Proof General

Organize your proofs!

User Manual for Proof General 4.0
October 2010
proofgeneral.inf.ed.ac.uk

David Aspinall and Thomas Kleymann
with P. Courtieu, H. Goguen, D. Sequeira, M. Wenzel.

This manual and the program Proof General are Copyright © 1998-2010 Proof General team, LFCS Edinburgh.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

This manual documents Proof General, Version 4.0, for use with GNU Emacs 23.2 or later versions (subject to Emacs API changes). Proof General is distributed under the terms of the GNU General Public License (GPL); please check the accompanying file 'COPYING' for more details.

Visit Proof General on the web at http://proofgeneral.inf.ed.ac.uk

ProofGeneral.texi,v 11.2 2010/10/11 00:36:55 da Exp

Preface

Welcome to Proof General!

This preface has some news about the current release, future plans, and acknowledgements to those who have helped along the way. The appendix [History of Proof General], page 67 contains old news about previous releases, and notes on the development of Proof General.

Proof General has a home page at `http://proofgeneral.inf.ed.ac.uk`. Visit this page for the latest version of this manual, other documentation, system downloads, etc.

News for Version 4.0

Proof General version 4.0 is a major overhaul of Proof General. The main changes are:

- support for GNU Emacs only, **you cannot use XEmacs any more**;
- a new **Unicode Tokens** mode, which now replaces X-Symbol, see Chapter 4 [Unicode symbols and special layout support], page 25;
- to allow "document centred" working, annotating scripts with prover output and automatically sending commands to the prover, see Section 3.1 [Document centred working], page 19;
- support for latest versions of provers (Isabelle2009-2 and Coq 8.2);
- numerous smaller enhancements and efficiency improvements.

See the '`CHANGES`' file in the distribution for more complete details of changes, and the appendix [History of Proof General], page 67 for old news.

Future

The aim of the Proof General project is to provide powerful environments and tools for interactive proof.

Proof General has been Emacs based so far and uses heavy per-prover customisation. The **Proof General Kit** project proposes that proof assistants use a *standard* XML-based protocol for interactive proof, dubbed **PGIP**. PGIP will enable middleware for interactive proof tools and interface components. Rather than configuring Proof General for your proof assistant, you will need to configure your proof assistant to understand PGIP. There is a similarity however; the design of PGIP was based heavily on the Emacs Proof General framework.

At the time of writing, the Proof General Kit software is in a prototype stage and the PGIP protocol is still being refined. We have a prototype Proof General plugin for the Eclipse IDE and a prototype version of a PGIP-enabled Isabelle. There is also a middleware component for co-ordinating proof written in Haskell, the *Proof General Broker*. Further collaborations are sought for more developments, especially the PGIP enabling of other provers. For more details, see the Proof General Kit webpage. Help us to help you organize your proofs!

Credits

The original developers of the basis of Proof General were:

- **David Aspinall**,
- **Healfdene Goguen**,
- **Thomas Kleymann**, and
- **Dilip Sequeira**.

LEGO Proof General (the successor of `lego-mode`) was written by Thomas Kleymann and Dilip Sequeira. It is no longer maintained. Coq Proof General was written by Healfdene Goguen, with

later contributions from Patrick Loiseleur. It is now maintained by Pierre Courtieu. Isabelle Proof General was written and is being maintained by David Aspinall. It has benefited greatly from tweaks and suggestions by Markus Wenzel, who wrote the first support for Isar and added Proof General support inside Isabelle. David von Oheimb supplied the original patches for X-Symbol support, which improved Proof General significantly. Christoph Wedler, the author of X-Symbol, provided much useful support in adapting his package for PG.

The generic base for Proof General was developed by Kleymann, Sequeira, Goguen and Aspinall. It follows some of the ideas used in Project CROAP. The project to implement a proof mode for LEGO was initiated in 1994 and coordinated until October 1998 by Thomas Kleymann, becoming generic along the way. In October 1998, the project became Proof General and has been managed by David Aspinall since then.

This manual was written by David Aspinall and Thomas Kleymann. Some words found their way here from the user documentation of LEGO mode, prepared by Dilip Sequeira. Healfdene Goguen supplied some text for Coq Proof General. Since Proof General 2.0, this manual has been maintained and improved by David Aspinall. Pierre Courtieu and Markus Wenzel have contributed some sections.

The Proof General project has benefited from (indirect) funding by EPSRC (*Applications of a Type Theory Based Proof Assistant* in the late 1990s and *The Integration and Interaction of Multiple Mathematical Reasoning Processes*, EP/E005713/1 (RA0084) in 2006-8), the EC (the Co-ordination Action *Types* and previous related projects), and the support of the LFCS. Version 3.1 was prepared whilst David Aspinall was visiting ETL, Japan, supported by the British Council.

For Proof General 3.7, Graham Dutton assisted with the project management system and web pages. Since then intermittent support has been provided by the Research and Teaching Services Unit of the computing support team at the School of Informatics. For testing and feedback for older versions of Proof General, thanks go to Rod Burstall, Martin Hofmann, and James McKinna, and some of those who continued to help with the latest 3.x series, named next.

For the Proof General 4.0 release, special thanks go to Stefan Monnier for patches and suggestions, to Makarius for many bug reports and help with Isabelle support and to Pierre Courtieu for providing new features for Coq support.

During the development of Proof General 3.x and 4.0 releases, many people helped provide testing and other feedback, including the Proof General maintainers, Paul Callaghan, Pierre Courtieu, and Markus Wenzel, Stefan Berghofer, Gerwin Klein, and other folk who tested prereleases or sent bug reports and patches, including Cuihtlauac Alvarado, Lennart Beringer, Pascal Brisset, James Brotherston, Martin Buechi, Pierre Casteran, Lucas Dixon, Erik Martin-Dorel, Matt Fairtlough, Ivan Filippenko, Georges Gonthier, Robin Green, Florian Haftmann, Kim Hyung Ho, Mark A. Hillebrand, Greg O'Keefe, Alex Krauss, Pierre Lescanne, John Longley, Erik Martin-Dorel, Assia Mahboubi, Adam Megacz, Stefan Monnier, Tobias Nipkow, Leonor Prensa Nieto, David von Oheimb, Lawrence Paulson, Paul Roziere, Randy Pollack, Robert R. Schneck, Norbert Schirmer, Sebastian Skalberg, Mike Squire, Norbert Voelker, Tjark Weber, Mitsuharu Yamamoto.

Thanks to all of you (and apologies to anyone missed).

1 Introducing Proof General

Proof General is a generic Emacs interface for interactive proof assistants,[1] developed at the LFCS in the University of Edinburgh.

You do not have to be an Emacs militant to use Proof General!

The interface is designed to be very easy to use. You develop your proof script[2] in-place rather than line-by-line and later reassembling the pieces. Proof General keeps track of which proof steps have been processed by the prover, and prevents you editing them accidently. You can undo steps as usual.

The aim of Proof General is to provide a powerful and configurable interface for numerous interactive proof assistants. We target Proof General mainly at intermediate or expert users, so that the interface should be useful for large proof developments.

Please help us!

Send us comments, suggestsions, or (the best) patches to improve support for your chosen proof assistant. Contact us at `http://proofgeneral.inf.ed.ac.uk/trac`.

If your chosen proof assistant isn't supported, read the accompanying *Adapting Proof General* manual to find out how to configure PG for a new prover.

1.1 Installing Proof General

If Proof General has not already been installed for you, you should unpack it and insert the line:

```
(load "proof-general-home/generic/proof-site.el")
```

into your '`~/.emacs`' file, where *proof-general-home* is the top-level directory that was created when Proof General was unpacked.

For much more information, See Appendix A [Obtaining and Installing], page 61.

1.2 Quick start guide

Once Proof General is correctly installed, the corresponding Proof General mode will be invoked automatically when you visit a proof script file for your proof assistant, for example:

Prover	Extensions	Mode
LEGO	'`.l`'	`lego-mode`
Coq	'`.v`'	`coq-mode`
Isabelle	'`.thy`'	`isar-mode`
Phox	'`.phx`'	`phox-mode`
HOL98	'`.sml`'	`hol98-mode`
ACL2	'`.acl2`'	`acl2-mode`
Twelf	'`.elf`'	`twelf-mode`
Plastic	'`.lf`'	`plastic-mode`
Lambda-CLAM	'`.lcm`'	`lclam-mode`
CCC	'`.ccc`'	`ccc-mode`
PG-Shell	'`.pgsh`'	`pgshell-mode`

(the exact list of Proof Assistants supported may vary according to the version of Proof General and its local configuration). You can also invoke the mode command directly, e.g., type *`M-x lego-mode`*, to turn a buffer into a lego script buffer.

You'll find commands to process the proof script are available from the toolbar, menus, and keyboard. Type *`C-h m`* to get a list of the keyboard shortcuts for the current mode. The

[1] A *proof assistant* is a computerized helper for developing mathematical proofs. For short, we sometimes call it a *prover*, although we always have in mind an interactive system rather than a fully automated theorem prover.

[2] A *proof script* is a sequence of commands which constructs a proof, usually stored in a file.

commands available should be easy to understand, but the rest of this manual describes them in some detail.

The proof assistant itself is started automatically inside Emacs as an "inferior" process when you ask for some of the proof script to be processed. You can start the proof assistant manually with the menu command "Start proof assistant".

To follow an example use of Proof General on a Isabelle proof, see Section 2.1 [Walkthrough example in Isabelle], page 7. If you know the syntax for proof scripts in another theorem prover, you can easily adapt the details given there.

1.3 Features of Proof General

Why would you want to use Proof General?

Proof General is designed to be useful for novices and expert users alike. It will be useful to you if you use a proof assistant, and you'd like an interface with the following features: simplified interaction, script management, multiple file scripting, a script editing mode, proof by pointing, toolbar and menus, syntax highlighting, real symbols, functions menu, tags, and finally, adaptability.

Here is an outline of some of these features. Look in the contents page or index of this manual to find out about the others!

- *Simplified interaction*
  Proof General is designed for proof assistants which have a command-line shell interpreter. When using Proof General, the proof assistant's shell is hidden from the user. Communication takes place via three buffers (Emacs text widgets). Communication takes place via three buffers. The *script buffer* holds input, the commands to construct a proof. The *goals buffer* displays the current list of subgoals to be solved. The *response buffer* displays other output from the proof assistant. By default, only two of these three buffers are displayed. This means that the user normally only sees the output from the most recent interaction, rather than a screen full of output from the proof assistant.

  Proof General does not commandeer the proof assistant shell: the user still has complete access to it if necessary.

  For more details, see Section 2.4 [Summary of Proof General buffers], page 11 and see Section 7.3 [Display customization], page 38.
- *Script management*
  Proof General colours proof script regions blue when they have been processed by the prover, and colours regions red when the prover is currently processing them. The appearance of Emacs buffers always matches the proof assistant's state. Coloured parts of the buffer cannot be edited. Proof General has functions for *asserting* or *retracting* parts of a proof script, which alters the coloured regions.

  For more details, see Chapter 2 [Basic Script Management], page 7, Section 2.6 [Script processing commands], page 13, and Chapter 3 [Advanced Script Management and Editing], page 19.
- *Script editing mode*
  Proof General provides useful facilities for editing proof scripts, including syntax hilighting and a menu to jump to particular goals, definitions, or declarations. Special editing functions send lines of proof script to the proof assistant, or undo previous proof steps.

  For more details, see Section 2.5 [Script editing commands], page 12, and Section 2.6 [Script processing commands], page 13.
- *Toolbar and menus*
  A script buffer has a toolbar with navigation buttons for processing parts of the proof

script. A menu provides further functions for operations in the proof assistant, as well as customization of Proof General.

For more details, see Section 2.8 [Toolbar commands], page 16, Section 2.7 [Proof assistant commands], page 14, and Chapter 7 [Customizing Proof General], page 37.

- *Proof by pointing*
  Proof General has support for proof-by-pointing and similar features. Proof by pointing allows you to click on a subterm of a goal to be proved, and automatically apply an appropriate proof rule or tactic. Proof by pointing is specific to the proof assistant (and logic) in use; therefore it is configured mainly on the proof assistant side. If you would like to see proof by pointing support for Proof General in a particular proof assistant, petition the developers of the proof assistant to provide it.

1.4 Supported proof assistants

Proof General comes ready-customized for several proof assistants, including these:

- **LEGO Proof General** for LEGO Version 1.3.1
  See Chapter 9 [LEGO Proof General], page 49, for more details.
- **Coq Proof General** for Coq Version 8.2
  See Chapter 10 [Coq Proof General], page 51, for more details.
- **Isabelle Proof General** for Isabelle2009-2
  See Chapter 11 [Isabelle Proof General], page 53, and documentation supplied with Isabelle for more details.
- **HOL Proof General** for HOL98 (HOL4)
  See Chapter 12 [HOL Proof General], page 57, for more details.
- **Shell Proof General** for shell scripts (not really a proof assistant!)
  See Chapter 13 [Shell Proof General], page 59, for more details.

Proof General is designed to be generic, so if you know how to write regular expressions, you can make:

- **Your Proof General** for your favourite proof assistant.
  For more details of how to make Proof General work with another proof assistant, see the accompanying manual *Adapting Proof General*.

The exact list of Proof Assistants supported may vary according to the version of Proof General you have and its local configuration; only the standard instances documented in this manual are listed above.

Note that there is some variation between the features supported by different instances of Proof General. The main variation is proof by pointing, which is only supported in LEGO at the moment. For advanced features like this, some extensions to the output routines of the proof assistant are required, typically. If you like Proof General, **please help us by asking the implementors of your favourite proof assistant to support Proof General** as much as possible.

1.5 Prerequisites for this manual

This manual assumes that you understand a little about using Emacs, for example, switching between buffers using `C-x b` and understanding that a key sequence like `C-x b` means "control with x, followed by b". A key sequence like `M-z` means "meta with z". (META may be labelled ALT on your keyboard).

The manual also assumes you have a basic understanding of your proof assistant and the language and files it uses for proof scripts. But even without this, Proof General is not useless: you can use the interface to *replay* proof scripts for any proof assistant without knowing how to start it up or issue commands, etc. This is the beauty of a common interface mechanism.

To get more from Proof General and adapt it to your liking, it helps to know a little bit about how Emacs lisp packages can be customized via the Customization mechanism. It's really easy to use. For details, see Section 7.2 [How to customize], page 37. See Info file '`(emacs)`', node '`Customization`', for documentation in Emacs.

To get the absolute most from Proof General, to improve it or to adapt it for new provers, you'll need to know a little bit of Emacs lisp. Emacs is self-documenting, so you can begin from `C-h` and find out everything! Here are some useful commands:

`C-h i`	`info`
`C-h m`	`describe-mode`
`C-h b`	`describe-bindings`
`C-h f`	`describe-function`
`C-h v`	`describe-variable`

1.6 Organization of this manual

This manual covers the user-level view and customization of Proof General. The accompanying *Adapting Proof General* manual considers adapting Proof General to new proof assistants, and documents some of the internals of Proof General.

Three appendices of this manual contain some details about obtaining and installing Proof General and some known bugs. The contents of these final chapters is also covered in the files '`INSTALL`' and '`BUGS`' contained in the distribution. Refer to those files for the latest information.

The manual concludes with some references and indexes. See the table of contents for full details.

2 Basic Script Management

This chapter is an introduction to using the script management facilities of Proof General. We begin with a quick walkthrough example, then describe the concepts and functions in more detail.

2.1 Walkthrough example in Isabelle

Here's a short example in Isabelle to see how script management is used. The file you are asked to type below is included in the distribution as '`isar/Example.thy`'. If you're not using Isabelle, substitute some lines from a simple proof for your proof assistant, or consult the example file supplied with Proof General for your prover, called something like '`foo/example.foo`' for a proof assistant Foo.

This walkthrough is keyboard based, but you could easily use the toolbar and menu functions instead. The best way to learn Emacs key bindings is by using the menus. You'll find the keys named below listed on the menus.

- First, start Emacs with Proof General loaded. According to how you have installed Proof General, this may be by typing `proofgeneral`, selecting it from a menu, or simply by starting Emacs itself.
- Next, find a new file by *C-x C-f* and typing as the filename '`Walkthrough.thy`'. This should load Isabelle Proof General and the toolbar and Proof General menus will appear. You should have an empty buffer displayed.

The notation *C-x C-f* means control key with 'x' followed by control key with 'f'. This is a standard notation for Emacs key bindings, used throughout this manual. This function also appears on the `File` menu of Emacs. The remaining commands used will be on the `Proof-General` menu.

If you're not using Isabelle, you must choose a different file extension, appropriately for your proof assistant. If you don't know what to use, see the previous chapter for the list of supported assistants and file extensions.

- Turn on *electric terminator* by typing *C-c ;* and enter:

  ```
  theory Walkthrough imports Main begin;
  ```

  This first command begins the definition of a new theory inside Isabelle, which extends the theory `Main`. (We're assuming that you have Isabelle/HOL available, which declares the `Main` theory. You should be able to see the list of installed logics in Isabelle on the `Logics` menu).

Electric terminator sends commands to the proof assistant as you type them. At the moment you type the semicolon, the `theory` command will be sent to Isabelle behind the scenes. First, there is a short delay while Isabelle is launched; you may see a welcome message. Then, you may notice that the command briefly is given an orange/pink background (or shown in inverse video if you don't have a colour display), before you see a window containing text like this:

```
theory Walkthrough
```

which reflects the command just executed.

In this case of this first command, it is hard to see the orange/pink stage because the command is processed very quickly on most machines. But in general, processing commands can take an arbitrary amount of time (or not terminate at all). For this reason, Proof General maintains a queue of commands which are sent one-by-one from the proof script. As Isabelle successfully processes commands in the queue, they will turn from the orange/pink colour into blue (by default). The blue regions indicate text that has been read by the prover and should not be edited; for this reason it is made read-only, by default. We often call it *locked*.

Electric terminator mode is popular, but not enabled by default because of the principle of least surprise. Moreover, in Isabelle, the semicolon terminators are optional so proof scripts are usually written without them to avoid clutter. You'll notice that although you typed a semi-colon it was not included in the buffer! The electric terminator tries to be smart about comments and strings; in the one case when you are trying to add a semi-colon inside an already written comment, you need to use *C-q ;* to quote the character.

Without electric terminator, you can trigger processing the text up to the current position of the point with the key *C-c C-RET*, or just up to the next command with *C-c C-n*. We show the rest of the example in Isabelle with semi-colons, but you can miss them out.

Coq, on the other hand, requires a full-stop terminator at the end of each line, so *C-c C-.* is the key binding used to turn on electric terminator. If you don't know what the terminator character is, you can find the option anyway on the menu: `Proof General -> Quick Options -> Processing -> Electric Terminator` which also shows the key binding.

If you want to use electric terminator, you can customize Proof General to enable it everytime if you want, See Chapter 7 [Customizing Proof General], page 37. For the common options, customization is easy: just use the menu item `Proof General -> Quick Options` to make your choices, and `Proof General -> Quick Options -> Save Options` to save your choices.

- Next type on a new line:

```
theorem my_theorem: "A & B --> B & A";
```

The goal we have set ourselves to prove should be displayed in the *goals buffer*.

- Now type:

```
proof;
  assume "A & C";
```

This will update the goals buffer.

But whoops! That was the wrong command, we typed `C` instead of `B`.

- Press *C-c C-BS* to pretend that didn't happen.

Note: *BS* means the backspace key. This key press sends an undo command to Isabelle, and deletes the `assume` command from the proof script. If you just want to undo without deleting, you can type *C-c C-u* instead, or use the toolbar navigation button.

- Instead, let's try:

```
assume "A & B";
```

  Which is better.

- From this assumption we can get `B` and `A` by the trivial step `..` which splits the assumption using an elimination step:

```
then obtain B and A ..;
```

- Finally, we establish the goal by the trivial step `..` again, which triggers an introduction rule:

```
then show "B & A" ..;
```

After this proof step, the message from Isabelle indicates that the proof has succeeded, so we can conclude the proof with the `qed` command.

- Finally, type:

```
qed;
```

This last command closes the proof and saves the proved theorem.

Moving the mouse pointer over the qed command now reveals that the entire proof has been aggregated into a single segment (if you did this before, you would see highlighting of each command separately). This reflects the fact that Isabelle has thrown away the history of the proof, so if we want to undo now, the whole proof must be retracted.

- Suppose we decide to call the theorem something more sensible. Move the cursor up into the locked region, somewhere between '`theorem`' and '`qed`', enter *C-c C-RET*.

You see that the locked segment for the whole proof is now unlocked (and uncoloured): it is transferred back into the editing region.

The command *C-c C-RET* moves the end of the locked region to the cursor position, or as near as possible above or below it, sending undoing commands or proof commands as necessary. In this case, the locked region will always be moved back to the end of the `theory` line, since that is the closest possible position to the cursor that appears before it. If you simply want to *retract* the whole file in one go, you can use the key *C-c C-r* (which corresponds to the up arrow on the toolbar), which will automatically move the cursor to the top of the file.

- Now improve the goal name, for example:

  ```
  theorem and_commutes: "A & B --> B & A"
  ```

  You can swiftly replay the rest of the buffer now with *C-c C-b* (or the down arrow on the toolbar).
- At the end of the buffer, you may insert the command

  ```
  end
  ```

  to complete the theory.

Notice that if you right-click on one of the highlighted regions in the blue area you will see a context menu for the region. This includes a "show/hide" option for *folding* a proof, as well as some editing commands for copying the region or rearranging its order in the processed text: "move up/move down". (These latter commands occasionally help you reorder text without needing to reprove it, although they risk breaking the proof!)

Finally, once you are happy with your theory, you should save the file with *C-x C-s* before moving on to edit another file or exiting Emacs. If you forget to do this, Proof General or Emacs will surely prompt you sooner or later!

2.2 Proof scripts

A *proof script* is a sequence of commands which constructs definitions, declarations, theories, and proofs in a proof assistant. Proof General is designed to work with text-based *interactive* proof assistants, where the mode of working is usually a dialogue between the human and the proof assistant.

Primitive interfaces for proof assistants simply present a *shell* (command interpreter) view of this dialogue: the human repeatedly types commands to the shell until the proof is completed. The system responds at each step, perhaps with a new list of subgoals to be solved, or perhaps with a failure report. Proof General manages the dialogue to show the human only the information which is relevant at each step.

Often we want to keep a record of the proof commands used to prove a theorem, to build up a library of proved results. An easy way to store a proof is to keep a text file which contains a proof script; proof assistants usually provide facilities to read a proof script from a file instead of the terminal. Using the file, we can *replay* the proof script to prove the theorem again.

Using only a primitive shell interface, it can be tedious to construct proof scripts with cut-and-paste. Proof General helps out by issuing commands directly from a proof script file, while it is being written and edited. Proof General can also be used conveniently to replay a proof step-by-step, to see the progress at each stage.

Scripting is the process of building up a proof script file or replaying a proof. When scripting, Proof General sends proof commands to the proof assistant one at a time, and prevents you from editing commands which have been successfully completed by the proof assistant, to keep synchronization. Regions of the proof script are analysed based on their syntax and the behaviour of the proof assistant after each proof command.

2.3 Script buffers

A *script buffer* is a buffer displaying a proof script. Its Emacs mode is particular to the proof assistant you are using (but it inherits from *proof-mode*).

A script buffer is divided into three regions: *locked*, *queue* and *editing*. The proof commands in the script buffer can include a number of *Goal-save sequences*.

2.3.1 Locked, queue, and editing regions

The three regions that a script buffer is divided into are:

- The *locked* region, which appears in blue (underlined on monochrome displays) and contains commands which have been sent to the proof process and verified. The commands in the locked region cannot be edited.
- The *queue* region, which appears in pink (inverse video) and contains commands waiting to be sent to the proof process. Like those in the locked region, these commands can't be edited.
- The *editing* region, which contains the commands the user is working on, and can be edited as normal Emacs text.

These three regions appear in the buffer in the order above; that is, the locked region is always at the start of the buffer, and the editing region always at the end. The queue region only exists if there is input waiting to be processed by the proof process.

Proof General has two fundamental operations which transfer commands between these regions: *assertion* (or processing) and *retraction* (or undoing).

Assertion causes commands from the editing region to be transferred to the queue region and sent one by one to the proof process. If the command is accepted, it is transferred to the locked region, but if an error occurs it is signalled to the user, and the offending command is transferred back to the editing region together with any remaining commands in the queue.

Assertion corresponds to processing proof commands, and makes the locked region grow.

Retraction causes commands to be transferred from the locked region to the editing region (again via the queue region) and the appropriate 'undo' commands to be sent to the proof process.

Retraction corresponds to undoing commands, and makes the locked region shrink. For details of the commands available for doing assertion and retraction, See Section 2.6 [Script processing commands], page 13.

2.3.2 Goal-save sequences

A proof script contains a sequence of commands used to prove one or more theorems.

As commands in a proof script are transferred to the locked region, they are aggregated into segments which constitute the smallest units which can be undone. Typically a segment consists of a declaration or definition, or all the text from a *goal* command to the corresponding *save* command, or the individual commands in the proof of an unfinished goal. As the mouse moves over the the region, the segment containing the pointer will be highlighted.

Proof General therefore assumes that the proof script has a series of proofs which look something like this:

```
goal mythm is G
...
save theorem mythm
```

interspersed with comments, definitions, and the like. Of course, the exact syntax and terminology will depend on the proof assistant you use.

The name *mythm* can appear in a menu for the proof script to help quickly find a proof (see Section 5.2 [Imenu and Speedbar], page 31).

2.3.3 Active scripting buffer

You can edit as many script buffers as you want simultaneously, but only one buffer at a time can be used to process a proof script incrementally: this is the *active scripting buffer*.

The active scripting buffer has a special indicator: the word `Scripting` appears in its mode line.

When you use a scripting command, it will automatically turn a buffer into the active scripting mode. You can also do this by hand, via the menu command 'Toggle Scripting' or the key *C-c C-s*.

C-c C-s `proof-toggle-active-scripting`

When active scripting mode is turned on, several things may happen to get ready for scripting (exactly what happens depends on which proof assistant you are using and some user settings). First, the proof assistant is started if it is not already running. Second, a command is sent to the proof assistant to change directory to the directory of the current buffer. If the current buffer corresponds to a file, this is the directory the file lives in. This is in case any scripting commands refer to files in the same directory as the script. The third thing that may happen is that you are prompted to save some unsaved buffers. This is in case any scripting commands may read in files which you are editing. Finally, some proof assistants may automatically read in files which the current file depends on implicitly. In Isabelle, for example, there is an implicit dependency between a `.ML` script file and a `.thy` theory file which defines its theory.

If you have a partly processed scripting buffer and use *C-c C-s*, or you attempt to use script processing in a new buffer, Proof General will ask you if you want to retract what has been proved so far, `Scripting incomplete in buffer myproof.l, retract?` or if you want to process the remainder of the active buffer, `Completely process buffer myproof.l instead?` before you can start scripting in a new buffer. If you refuse to do either, Proof General will give an error message: `Cannot have more than one active scripting buffer!`.

To turn off active scripting, the buffer must be completely processed (all blue), or completely unprocessed. There are two reasons for this. First, it would certainly be confusing if it were possible to split parts of a proof arbitrarily between different buffers; the dependency between the commands would be lost and it would be tricky to replay the proof.[1] Second, we want to interface with file management in the proof assistant. Proof General assumes that a proof assistant may have a notion of which files have been processed, but that it will only record files that have been *completely* processed. For more explanation of the handling of multiple files, See Section 3.4 [Switching between proof scripts], page 21.

`proof-toggle-active-scripting` **&optional** *arg* [Command]
Toggle active scripting mode in the current buffer.
With *arg*, turn on scripting iff *arg* is positive.

2.4 Summary of Proof General buffers

Proof General manages several kinds of buffers in Emacs. Here is a summary of the different kinds of buffers you will use when developing proofs.

- The *proof shell buffer* is an Emacs shell buffer used to run your proof assistant. Usually it is hidden from view (but see Section 3.9 [Escaping script management], page 22). Communication with the proof shell takes place via two or three intermediate buffers.
- A *script buffer*, as we have explained, is a buffer for editing a proof script. The *active scripting buffer* is the script buffer which is currently being used to send commands to the proof shell.

[1] Some proof assistants provide some level of support for switching between multiple concurrent proofs, but Proof General does not use this. Generally the exact context for such proofs is hard to define to easily split them into multiple files.

- The *goals buffer* displays the list of subgoals to be solved for a proof in progress. During a proof it is usually displayed together with the script buffer. The goals buffer has facility for *proof-by-pointing*.
- The *response buffer* displays other output from the proof assistant, for example error messages or informative messages. The response buffer is displayed whenever Proof General puts a new message in it.
- The *trace buffer* is a special version of the response buffer. It may be used to display unusual debugging output from the prover, for example, tracing proof tactics or rewriting procedures. This buffer is also displayed whenever Proof General puts a new message in it (although it may be quickly replaced with the response or goals buffer in two-buffer mode).

Normally Proof General will automatically reveal and hide the goals and response buffers as necessary during scripting. However there are ways to customize the way the buffers are displayed, for example, to prevent auxiliary buffers being displayed at all (see Section 7.3 [Display customization], page 38).

The menu `Proof General -> Buffers` provides a convenient way to display or switch to a Proof General buffer: the active scripting buffer; the goal or response buffer; the tracing buffer; or the shell buffer. Another command on this menu, `Clear Responses`, clears the response and tracing buffer.

2.5 Script editing commands

Proof General provides a few functions for editing proof scripts. The generic functions mainly consist of commands to navigate within the script. Specific proof assistant code may add more to these basics.

Indentation is controlled by the user option `proof-script-indent` (see Section 7.4 [User options], page 40). When indentation is enabled, Proof General will indent lines of proof script with the usual Emacs functions, particularly *TAB*, `indent-for-tab-command`. Unfortunately, indentation in Proof General 4.0 is somewhat slow. Therefore with large proof scripts, we recommend `proof-script-indent` is turned off.

Here are the commands for moving around in a proof script, with their default key-bindings:

C-c C-a `proof-goto-command-start`

C-c C-e `proof-goto-command-end`

C-c C-. `proof-goto-end-of-locked`

`proof-goto-command-start` [Command]
Move point to start of current (or final) command of the script.

`proof-goto-command-end` [Command]
Set point to end of command at point.

The variable `proof-terminal-string` is a prover-specific string to terminate proof commands. LEGO and Isabelle use a semicolon, '`;`'. Coq employs a full-stop '`.`'.

`proof-goto-end-of-locked` **&optional** *switch* [Command]
Jump to the end of the locked region, maybe switching to script buffer.
If called interactively or *switch* is non-nil, switch to script buffer. If called interactively, a mark is set at the current location with '`push-mark`'

2.6 Script processing commands

Here are the commands for asserting and retracting portions of the proof script, together with their default key-bindings. Sometimes assertion and retraction commands can only be issued when the queue is empty. You will get an error message `Proof Process Busy!` if you try to assert or retract when the queue is being processed.[2]

C-c C-n `proof-assert-next-command-interactive`

C-c C-u `proof-undo-last-successful-command`

C-c C-BS `proof-undo-and-delete-successful-command`

C-c C-RET `proof-goto-point`

C-c C-b `proof-process-buffer`

C-c C-r `proof-retract-buffer`

C-c terminator-character
`proof-electric-terminator-toggle`

The last command, `proof-electric-terminator-toggle`, is triggered using the character which terminates proof commands for your proof assistant's script language. For LEGO and Isabelle, use *C-c ;*, for Coq, use *C-c .*. This not really a script processing command. Instead, if enabled, it causes subsequent key presses of *;* or *.* to automatically activate `proof-assert-next-command-interactive` for convenience.

Rather than use a file command inside the proof assistant to read a proof script, a good reason to use *C-c C-b* (`proof-process-buffer`) is that with a faulty proof script (e.g., a script you are adapting to prove a different theorem), Proof General will stop exactly where the proof script fails, showing you the error message and the last processed command. So you can easily continue development from exactly the right place in the script.

Here is the full set of script processing commands.

`proof-assert-next-command-interactive` [Command]
Process until the end of the next unprocessed command after point.
If inside a comment, just process until the start of the comment.

`proof-undo-last-successful-command` [Command]
Undo last successful command at end of locked region.

`proof-undo-and-delete-last-successful-command` [Command]
Undo and delete last successful command at end of locked region.
Useful if you typed completely the wrong command. Also handy for proof by pointing, in case the last proof-by-pointing command took the proof in a direction you don't like.

Notice that the deleted command is put into the Emacs kill ring, so you can use the usual '`yank`' and similar commands to retrieve the deleted text.

`proof-goto-point` [Command]
Assert or retract to the command at current position.
Calls '`proof-assert-until-point`' or '`proof-retract-until-point`' as appropriate.

`proof-process-buffer` [Command]
Process the current (or script) buffer, and maybe move point to the end.

[2] In fact, this is an unnecessary restriction imposed by the original design of Proof General. There is nothing to stop future versions of Proof General allowing the queue region to be extended or shrunk, whilst the prover is processing it. Proof General 3.0 already relaxes the original design, by allowing successive assertion commands without complaining.

`proof-retract-buffer` [Command]
Retract the current buffer, and maybe move point to the start.

`proof-electric-terminator-toggle` **&optional** *arg* [Command]
Toggle '`proof-electric-terminator-enable`'. With *arg*, turn on iff ARG>0.
This function simply uses `customize-set-variable` to set the variable.

`proof-assert-until-point-interactive` [Command]
Process the region from the end of the locked-region until point.
If inside a comment, just process until the start of the comment.

`proof-retract-until-point-interactive` **&optional** *delete-region* [Command]
Tell the proof process to retract until point.
If invoked outside a locked region, undo the last successfully processed command. If called with a prefix argument (*delete-region* non-nil), also delete the retracted region from the proof-script.

As experienced Emacs users will know, a *prefix argument* is a numeric argument supplied by some key sequence typed before a command key sequence. You can supply a specific number by typing META with the digits, or a "universal" prefix of `C-u`. See See Info file '`(emacs)`', node '`Arguments`' for more details. Several Proof General commands, like `proof-retract-until-point-interactive`, may accept a *prefix argument* to adjust their behaviour somehow.

2.7 Proof assistant commands

There are several commands for interacting with the proof assistant and Proof General, which do not involve the proof script. Here are the key-bindings and functions.

`C-c C-l`	`proof-display-some-buffers`
`C-c C-p`	`proof-prf`
`C-c C-t`	`proof-ctxt`
`C-c C-h`	`proof-help`
`C-c C-i`	`proof-query-identifier`
`C-c C-f`	`proof-find-theorems`
`C-c C-w`	`pg-response-clear-displays`
`C-c C-c`	`proof-interrupt-process`
`C-c C-v`	`proof-minibuffer-cmd`
`C-c C-s`	`proof-shell-start`
`C-c C-x`	`proof-shell-exit`

`proof-display-some-buffers` [Command]
Display the reponse, trace, goals, or shell buffer, rotating.
A fixed number of repetitions of this command switches back to the same buffer. Also move point to the end of the response buffer if it's selected. If in three window or multiple frame mode, display two buffers. The idea of this function is to change the window->buffer mapping without adjusting window layout.

`proof-prf` [Command]
Show the current proof state.
Issues a command to the assistant based on `proof-showproof-command`.

`proof-ctxt` [Command]
Show the current context.
Issues a command to the assistant based on `proof-context-command`.

`proof-help` [Command]
Show a help or information message from the proof assistant.
Typically, a list of syntax of commands available. Issues a command to the assistant based on `proof-info-command`.

`proof-query-identifier` *string* [Command]
Query the prover about the identifier *string*.
If called interactively, *string* defaults to the current word near point.

`proof-find-theorems` *arg* [Command]
Search for items containing given constants.
Issues a command based on *arg* to the assistant, using `proof-find-theorems-command`. The user is prompted for an argument.

`pg-response-clear-displays` [Command]
Clear Proof General response and tracing buffers.
You can use this command to clear the output from these buffers when it becomes overly long. Particularly useful when '`proof-tidy-response`' is set to nil, so responses are not cleared automatically.

`proof-interrupt-process` [Command]
Interrupt the proof assistant. Warning! This may confuse Proof General.

This sends an interrupt signal to the proof assistant, if Proof General thinks it is busy.

This command is risky because we don't know whether the last command succeeded or not. The assumption is that it didn't, which should be true most of the time, and all of the time if the proof assistant has a careful handling of interrupt signals.

Some provers may ignore (and lose) interrupt signals, or fail to indicate that they have been acted upon yet stop in the middle of output. In the first case, PG will terminate the queue of commands at the first available point. In the second case, you may need to press enter inside the prover command buffer (e.g., with Isabelle*2009* press RET inside **isabelle**).

`proof-minibuffer-cmd` *cmd* [Command]
Send *cmd* to proof assistant. Interactively, read from minibuffer.
The command isn't added to the locked region.

If a prefix arg is given and there is a selected region, that is pasted into the command. This is handy for copying terms, etc from the script.

If '`proof-strict-state-preserving`' is set, and '`proof-state-preserving-p`' is configured, then the latter is used as a check that the command will be safe to execute, in other words, that it won't ruin synchronization. If when applied to the command it returns false, then an error message is given.

warning: this command risks spoiling synchronization if the test '`proof-state-preserving-p`' is not configured, if it is only an approximate test, or if '`proof-strict-state-preserving`' is off (nil).

As if the last two commands weren't risky enough, there's also a command which explicitly adjusts the end of the locked region, to be used in extreme circumstances only. See Section 3.9 [Escaping script management], page 22.

There are a few commands for starting, stopping, and restarting the proof assistant process. The first two have key bindings but restart does not. As with any Emacs command, you can invoke these with `M-x` followed by the command name.

`proof-shell-start` [Command]
Initialise a shell-like buffer for a proof assistant.
Does nothing if proof assistant is already running.

Also generates goal and response buffers.

If '`proof-prog-name-ask`' is set, query the user for the process command.

`proof-shell-exit` [Command]
Query the user and exit the proof process.

This simply kills the '`proof-shell-buffer`' relying on the hook function '`proof-shell-kill-function`' to do the hard work.

`proof-shell-restart` [Command]
Clear script buffers and send '`proof-shell-restart-cmd`'.
All locked regions are cleared and the active scripting buffer deactivated.

If the proof shell is busy, an interrupt is sent with '`proof-interrupt-process`' and we wait until the process is ready.

The restart command should re-synchronize Proof General with the proof assistant, without actually exiting and restarting the proof assistant process.

It is up to the proof assistant how much context is cleared: for example, theories already loaded may be "cached" in some way, so that loading them the next time round only performs a re-linking operation, not full re-processing. (One way of caching is via object files, used by Lego and Coq).

2.8 Toolbar commands

The toolbar provides a selection of functions for asserting and retracting portions of the script, issuing non-scripting commands to inspect the prover's state, and inserting "goal" and "save" type commands. The latter functions are not available on keys, but are available from the from the menu, or via *M-x*, as well as the toolbar.

`proof-issue-goal` *arg* [Command]
Write a goal command in the script, prompting for the goal.
Issues a command based on *arg* to the assistant, using `proof-goal-command`. The user is prompted for an argument.

`proof-issue-save` *arg* [Command]
Write a save/qed command in the script, prompting for the theorem name.
Issues a command based on *arg* to the assistant, using `proof-save-command`. The user is prompted for an argument.

2.9 Interrupting during trace output

If your prover generates output which is recognized as tracing output in Proof General, you may need to know about a special provision for interrupting the prover process. If the trace output is voluminous, perhaps looping, it may be difficult to interrupt with the ordinary *C-c C-c* (`proof-interrupt-process`) or the corresponding button/menu. In this case, you should try Emacs's **quit key**, *C-g*. This will cause a quit in any current editing commands, as usual, but during tracing output it will also send an interrupt signal to the prover. Hopefully this will stop the tracing output, and Emacs should catch up after a short delay.

Here's an explanation of the reason for this special provision. When large volumes of output from the prover arrive quickly in Emacs, as typically is the case during tracing (especially tracing looping tactics!), Emacs may hog the CPU and spend all its time updating the display with

the trace output. This is especially the case when features like output fontification and token display are active. If this happens, ordinary user input in Emacs is not processed, and it becomes difficult to do normal editing. The root of the problem is that Emacs runs in a single thread, and pending process output is dealt with before pending user input. Whether or not you see this problem depends partly on the processing power of your machine (or CPU available to Emacs when the prover is running). One way to test is to start an Emacs shell with *M-x shell* and type a command such as `yes` which produces output indefinitely. Now see if you can interrupt the process! (Warning — on slower machines especially, this can cause lockups, so use a fresh Emacs.)

3 Advanced Script Management and Editing

If you are working with large proof developments, you may want to know about the advanced script management and editing features of Proof General covered in this chapter.

3.1 Document centred working

Proof scripts can be annotated with the output produced by the prover while they are checked. By hovering the mouse on the completed regions you can see any output that was produced when they were checked. Depending on the proof language (it works well with declarative languages), this may enable a "document centred" way of working, where you may not need to keep a separate window open for displaying prover output.

Several configuration settings can be used to fine tune this behaviour.

First, you may select

```
Proof General -> Quick Options -> Processing -> Full Annotations
```

to ensure that the details are recorded in the script. This is not the default because it can cause long sequences of commands to execute more slowly as the prover must output information. It also increases the space requirements for Emacs buffers. However, when interactively developing smaller files, it is very useful.

Next, you may *de*select

```
Proof General -> Quick Options -> Display -> Auto Raise
```

which will prevent the prover output being eagerly displayed. You can still manually arrange your Emacs windows and frames to ensure the output buffers are present if you want.

You may like to *de*select

```
Proof General -> Quick Options -> Display -> Colour Locked
```

to prevent highlighting of the locked region. This text which has been checked and that which has not is less obvious, but you can see the position of the next command to be processed with the marker.

If you have no colouring on the locked region, it can be hard to see where processing has got to. Look for the "overlay marker", a triangle in the left-hand fringe of the display, to see which line processing has stopped at. If it has stopped on a region with an error, you might want to see that. You can select

```
Proof General -> Quick Options -> Display -> Sticky Errors
```

to add a higlight for regions which did not successfully process on the last attempt. Whenever the region is edited, the highlight is removed.

Finally, you may want to ensure that

```
Proof General -> Quick Options -> Read Only -> Undo On Edit
```

is selected. Undo on edit is a setting for the 'proof-strict-read-only' variable. This allows you to freely edit the processed region, but first it automatically retracts back to the point of the edit. Comments can be edited freely without retraction.

See Section 7.3 [Display customization], page 38, for more information about customizing display options.

To help configure the options appropriately for document-centred working, there are two command

```
Proof General -> Quick Options -> Display -> Document Centred
Proof General -> Quick Options -> Display -> Default
```

which change settings appropriately between a document centred mode and the classic Proof General behaviour and appearance. The first command also engages automatic processing of the whole buffer, explained in the next section.

3.2 Automatic processing

If you like making your hair stand on end, the electric terminator mode is probably not enough. Proof General has another feature that will automatically send text to the prover, while you aren't looking.

Enabling

```
Proof General -> Quick Options -> Processing -> Process Automatically
```

Causes Proof General to start processing text when Emacs is idle for a while. You can choose either to send just the next command beyond the point, or the whole buffer. See

```
Proof General -> Quick Options -> Processing -> Automatic Processing Mode
```

for the choices.

The text will be sent in a fast loop that processes more quickly than `C-c C-b` (i.e., `proof-process-buffer`, the down toolbar button), but ignores user input and doesn't update the display. But the feature tries to be non-intrusive to the user: if you start to type something or use the mouse, the fast loop will be interrupted and revert to a slower interactive loop with display updates.

In the check next command mode, the successfully checked region will briefly flash up as green to indicate it is okay.

You can use `C-c C-.` (`proof-goto-end-of-locked`) to find out where processing got to, as usual. Text is only sent if the last interactive command processed some text (i.e., wasn't an undo step backwards into the buffer) and processing didn't stop with an error. To start automatic processing again after an error, simply hit `C-c C-n`. To turn the automatic processing on or off from the keyboard, you can use the keybinding:

`C-c >` `proof-autosend-toggle`

`proof-autosend-toggle` **&optional** *arg* [Command]
Toggle '`proof-autosend-enable`'. With *arg*, turn on iff ARG>0.
This function simply uses `customize-set-variable` to set the variable.

3.3 Visibility of completed proofs

Large developments may consist of large files with many proofs. To help see what has been proved without the detail of the proof itself, Proof General can hide portions of the proof script. Two different kinds of thing can be hidden: comments and (what Proof General designates as) the body of proofs.

You can toggle the visibility of a proof script portion by using the context sensitive menu triggered by **clicking the right mouse button on a completed proof**, or the key `C-c v`, which runs `pg-toggle-visibility`.

You can also select the "disappearing proofs" mode from the menu,

```
Proof-General -> Quick Options -> Display -> Disappearing Proofs
```

This automatically hides each the body of each proof portion as it is completed by the proof assistant. Two further menu commands in the main Proof-General menu, *Show all* and *Hide all* apply to all the completed portions in the buffer.

Notice that by design, this feature only applies to completed proofs, *after* they have been processed by the proof assistant. When files are first visited in Proof General, no information is stored about proof boundaries.

The relevant elisp functions and settings are mentioned below.

`pg-toggle-visibility` [Command]
Toggle visibility of region under point.

`pg-show-all-proofs` [Command]
Display all completed proofs in the buffer.

`pg-hide-all-proofs` [Command]
Hide all completed proofs in the buffer.

`proof-disappearing-proofs` [User Option]
Non-nil causes Proof General to hide proofs as they are completed.

The default value is `nil`.

3.4 Switching between proof scripts

Basic modularity in large proof developments can be achieved by splitting proof scripts across various files. Let's assume that you are in the middle of a proof development. You are working on a soundness proof of Hoare Logic in a file called[1] '`HSound.l`'. It depends on a number of other files which develop underlying concepts e.g. syntax and semantics of expressions, assertions, imperative programs. You notice that the current lemma is too difficult to prove because you have forgotten to prove some more basic properties about determinism of the programming language. Or perhaps a previous definition is too cumbersome or even wrong.

At this stage, you would like to visit the appropriate file, say '`sos.l`' and retract to where changes are required. Then, using script management, you want to develop some more basic theory in '`sos.l`'. Once this task has been completed (possibly involving retraction across even earlier files) and the new development has been asserted, you want to switch back to '`HSound.l`' and replay to the point you got stuck previously.

Some hours (or days) later you have completed the soundness proof and are ready to tackle new challenges. Perhaps, you want to prove a property that builds on soundness or you want to prove an orthogonal property such as completeness.

Proof General lets you do all of this while maintaining the consistency between proof script buffers and the state of the proof assistant. However, you cannot have more than one buffer where only a fraction of the proof script contains a locked region. Before you can employ script management in another proof script buffer, you must either fully assert or retract the current script buffer.

3.5 View of processed files

Proof General tries to be aware of all files that the proof assistant has processed or is currently processing. In the best case, it relies on the proof assistant explicitly telling it whenever it processes a new file which corresponds[2] to a file containing a proof script.

If the current proof script buffer depends on background material from other files, proof assistants typically process these files automatically. If you visit such a file, the whole file is locked as having been processed in a single step. From the user's point of view, you can only retract but not assert in this buffer. Furthermore, retraction is only possible to the *beginning* of the buffer.

Unlike a script buffer that has been processed step-by-step via Proof General, automatically loaded script buffers do not pass through a "red" phase to indicate that they are currently being processed. This is a limitation of the present implementation. Proof General locks a buffer as soon as it sees the appropriate message from the proof assistant. Different proof assistants may use different messages: either *early locking* when processing a file begins (e.g. LEGO) or *late locking* when processing a file ends (e.g. Isabelle).

---

[1] The suffix may depend of the specific proof assistant you are using e.g, LEGO's proof script files have to end with '`.l`'.

[2] For example, LEGO generates additional compiled (optimised) proof script files for efficiency.

With *early locking*, you may find that a script which has only been partly processed (due to an error or interrupt, for example), is wrongly completely locked by Proof General. Visit the file and retract back to the start to fix this.

With *late locking*, there is the chance that you can break synchronization by editing a file as it is being read by the proof assistant, and saving it before processing finishes.

In fact, there is a general problem of editing files which may be processed by the proof assistant automatically. Synchronization can be broken whenever you have unsaved changes in a proof script buffer and the proof assistant processes the corresponding file. (Of course, this problem is familiar from program development using separate editors and compilers). The good news is that Proof General can detect the problem and flashes up a warning in the response buffer. You can then visit the modified buffer, save it and retract to the beginning. Then you are back on track.

3.6 Retracting across files

Make sure that the current script buffer has either been completely asserted or retracted (Proof General enforces this). Then you can retract proof scripts in a different file. Simply visit a file that has been processed earlier and retract in it, using the retraction commands from see Section 2.6 [Script processing commands], page 13. Apart from removing parts of the locked region in this buffer, all files which depend on it will be retracted (and thus unlocked) automatically. Proof General reminds you that now is a good time to save any unmodified buffers.

3.7 Asserting across files

Make sure that the current script buffer has either been completely asserted or retracted. Then you can assert proof scripts in a different file. Simply visit a file that contains no locked region and assert some command with the usual assertion commands, see Section 2.6 [Script processing commands], page 13. Proof General reminds you that now is a good time to save any unmodified buffers. This is particularly useful as assertion may cause the proof assistant to automatically process other files.

3.8 Automatic multiple file handling

To make it easier to adapt Proof General for a proof assistant, there is another possibility for multiple file support — that it is provided automatically by Proof General and not integrated with the file-management system of the proof assistant.

In this case, Proof General assumes that the only files processed are the ones it has sent to the proof assistant itself. Moreover, it (conservatively) assumes that there is a linear dependency between files in the order they were processed.

If you only have automatic multiple file handling, you'll find that any files loaded directly by the proof assistant are *not* locked when you visit them in Proof General. Moreover, if you retract a file it may retract more than is strictly necessary (because it assumes a linear dependency).

For further technical details of the ways multiple file scripting is configured, see *Handling multiple files* in the *Adapting Proof General* manual.

3.9 Escaping script management

Occasionally you may want to review the dialogue of the entire session with the proof assistant, or check that it hasn't done something unexpected. Experienced users may also want to directly communicate with the proof assistant rather than sending commands via the minibuffer, see Section 2.7 [Proof assistant commands], page 14.

Although the proof shell is usually hidden from view, it is run in a buffer which you can use to interact with the prover if necessary. You can switch to it using the menu:

```
Proof-General -> Buffers -> Shell
```

Warning: you can probably cause confusion by typing in the shell buffer! Proof General may lose track of the state of the proof assistant. Output from the assistant is only fully monitored when Proof General is in control of the shell. When in control, Proof General watches the output from the proof assistant to guess when a file is loaded or when a proof step is taken or undone. What happens when you type in the shell buffer directly depends on how complete the communication is between Proof General and the prover (which depends on the particular instantiation of Proof General).

If synchronization is lost, you have two options to resynchronize. If you are lucky, it might suffice to use the key:

C-c C-z `proof-frob-locked-end`

This command is disabled by default, to protect novices using it accidently.

If `proof-frob-locked-end` does not work, you will need to restart script management altogether (see Section 2.7 [Proof assistant commands], page 14).

`proof-frob-locked-end` [Command]
Move the end of the locked region backwards to regain synchronization.
Only for use by consenting adults.

This command can be used to repair synchronization in case something goes wrong and you want to tell Proof General that the proof assistant has processed less of your script than Proof General thinks.

You should only use it to move the locked region to the end of a proof command.

3.10 Editing features

To make editing proof scripts more productive, Proof General provides some additional editing commands.

One facility is the *input ring* of previously processed commands. This allows a convenient way of repeating an earlier command or a small edit of it. The feature is reminiscent of history mechanisms provided in shell terminals (and the implementation is borrowed from the Emacs Comint package). The input ring only contains commands which have been successfully processed (coloured blue). Duplicated commands are only entered once. The size of the ring is set by the variable `pg-input-ring-size`.

M-p `pg-previous-matching-input-from-input`

M-n `pg-next-matching-input-from-input`

`pg-previous-input` *arg* [Command]
Cycle backwards through input history, saving input.

`pg-next-input` *arg* [Command]
Cycle forwards through input history.

`pg-previous-matching-input` *regexp n* [Command]
Search backwards through input history for match for *regexp*.
(Previous history elements are earlier commands.) With prefix argument *n*, search for Nth previous match. If *n* is negative, find the next or Nth next match.

`pg-next-matching-input` *regexp n* [Command]
Search forwards through input history for match for *regexp*.
(Later history elements are more recent commands.) With prefix argument *n*, search for Nth following match. If *n* is negative, find the previous or Nth previous match.

`pg-previous-matching-input-from-input` *n* [Command]
Search backwards through input history for match for current input.
(Previous history elements are earlier commands.) With prefix argument *n*, search for Nth previous match. If *n* is negative, search forwards for the -Nth following match.

`pg-next-matching-input-from-input` *n* [Command]
Search forwards through input history for match for current input.
(Following history elements are more recent commands.) With prefix argument *n*, search for Nth following match. If *n* is negative, search backwards for the -Nth previous match.

4 Unicode symbols and special layout support

Proof General inherits support for displaying Unicode (and any other) fonts from the underlying Emacs program. If you are lucky, your system will be able to use or synthesise a font that provides a rich set of mathematical symbols. To store symbols directly in files you need to use a particular coding, for example UTF-8. Newer Emacs versions can handle a multitude of different coding systems and will try to automatically detect an appropriate one; consult the Emacs documentation for more details. Of course, the prover that you are using will need to understand the same encodings and symbol meanings.

Alternatively, you can use the **Unicode Tokens** mode provided in Proof General to display mathematical symbols in place of sequences of other characters (usually plain ASCII). This can provide better compatibility, portability, and flexibility. Even if you use real Unicode characters as prover input, the Unicode Tokens mode can provide some helpful facilities for input shorthands and giving special layout.

4.1 Maths menu

The **Maths Menu** minor mode (adapted from a menu by Dave Love) simply adds a menu `Maths` to the main menubar for inserting common mathematical symbols. You can enable or disable it via the menu

```
Proof-General -> Quick Options -> Minor Modes -> Unicode Maths Menu
```

(`proof-maths-menu-toggle`). Whether or not the symbols display well the menus depends on the font used to display the menus (which depends on the Emacs version, toolkit and platform). Ordinarily, the symbols inserted into the text will be Unicode characters which will be saved in the file using the encoding selected by standard Emacs mechanisms.

4.2 Unicode Tokens mode

The **Unicode Tokens** minor mode has been written specially for Proof General (with thanks to Stefan Monnier for providing inspiration and a starting point). It supports the display of symbols when the underlying text of the file and buffer actually contains something else, typically, plain ASCII text. It provides backward compatibility with the older X-Symbol mode.

Unicode Tokens can be enabled or disabled using the menu:

```
Proof-General -> Quick Options -> Display -> Unicode Tokens
```

The mode to allows ASCII tokens (i.e., sequences of plain ASCII characters) to be displayed as Unicode character compositions, perhaps with additional text properties. The additional text properties allow the use of tokens to cause font changes (bold, italic), text size changes, and sub-script/super-script.

For example, the ASCII sequences `/\` or `\<And>` could be displayed as a conjunction symbol. The sequence `x __ y` might be written to display `y` as subscript. This allows a file to be stored in perfectly portable plain ASCII encoding, but be displayed and edited with real symbols and appealing layout. Of course, the proof assistant needs to understand the underlying tokens in each case.

Technically, the mechanism is based on Emacs Font Lock facility, using the `composition` text property to display ASCII character sequence tokens as something else. This means that the underlying buffer text is *not* altered. This is a major advantage over the older X-Symbol (and the experimental version of Unicode Tokens in PG 3.7.1), which had the annoying risk of saving your buffer text in a corrupted format. This can never happen with the new mode.

When the Unicode Tokens mode is enabled, Maths Menu is automatically modified to insert tokenised versions of the Unicode characters (whenever a reverse mapping can be found). This means that you can still use the Maths Menu to conveniently input symbols. You can easily

add custom key bindings for particular symbols you need to enter often (see Section 8.1 [Adding your own keybindings], page 45 for examples).

The Unicode Tokens mode also allows short-cut sequences of ordinary characters to quickly type tokens (similarly to the facility provided by X-Symbol). These, along with the token settings themselves, are configured on a per-prover basis.

4.3 Configuring tokens symbols and shortcuts

To edit the strings used to display tokens, or the collection of short-cuts, you can edit the file `PA-unicode-tokens.el`, or customize the main variables it contains, for example `PA-token-name-alist` and `PA-shortcut-alist`.

E.g., for Isabelle

```
M-x customize-variable isar-token-name-alist RET
```

provides an interface to the tokens, and

```
M-x customize-variable isar-shortcut-alist
```

an interface to the shortcuts.

Where possible, it is better to use the more fine grained way is available to do this, which edits the separate tables which are combine to form the big list of tokens. This is available via the menus, e.g., in Isabelle, use

```
Tokens -> Customize -> Extended Symbols
```

to customize the symbols used for the "extended" (non standard) symbol list.

4.4 Special layout

The Unicode Tokens mode supports both *symbol tokens* used to display character sequences in different ways and *control tokens* used to control the layout of the text in various ways, such as superscript, subscript, large, small, bold, italic, etc. (The size and position layout is managed using Emacs's `display` text property)

As well as displaying token sequences as special symbols, symbol tokens themselves can define layout options as well; for example you might define a token `\<hugeOplus>` to display a large circled-plus glyph. If you try the customization mentioned in the section above you will see the options available when defining symbols.

These options are fixed layout schemes which also make layout tokens easy to configure for provers. The layout possibilities include the ones shown in the table below. There are two ways of configuring control tokens for layout: *character controls* and *region controls*. The character controls apply to the next "character", although this is a prover-specific notion and might actually mean the next word or identifier. An example might be writing `BOLDCHAR x` to make a bold **x**. Similarly the region controls apply to a delineated region of text, for example, writing `BEGINBOLD this is bold ENDBOLD` could cause the enclosed text **this is bold** to be displayed in a bold font.

The control tokens that have been configured populate the Tokens menu, so, for example, you may be able to select a region of text and then use the menu item:

```
Tokens -> Format Region -> Bold
```

to cause the bold region tokens to be inserted around the selected text, which should cause the buffer presentation to show the text in a bold format (hiding the tokens).

Here is the table of layout controls available. What you actually can use will depend on the configuration for the underlying prover.

`sub` lower the text (subscript)

`sup` raise the text (superscript)

`bold` make the text be in the bold weight of the current font

`italic` make the text be in the italic variant of the current font

`big` make the text be in a bigger size of the current font

`small` make the text be in a smaller size of the current font

`underline`
underline the text

`overline` overline the text

`script` display the text in a "script" font

`frakt` display the text in a "fraktur" font

`serif` display the text in a serif font

`sans` display the text in a sans serif font

`keyword` display the text in the keyword face (`font-lock-keyword-face`)

`function` display the text in the function name face (`font-lock-function-name-face`)

`type` display the text in the type name face (`font-lock-type-face`)

`preprocessor`
display the text in the preprocessor face (`font-lock-preprocessor-face`)

`doc` display the text in the documentation face (`font-lock-doc-face`)

`builtin` display the text in the builtin face (`font-lock-builtin-face`)

Notice that the fonts can be set conveniently by the menu commands

```
Tokens -> Set Fonts -> Script
```

etc. See Section 4.7 [Selecting suitable fonts], page 29, for more.

The symbols used to select the various font-lock faces (see `M-x list-faces-display` to show them) allow you to define custom colouring of text for proof assistant input and output, exploiting rich underlying syntax mechanisms of the prover.

`unicode-tokens-serif-font-face` [Face]
Serif (roman) font face.

`unicode-tokens-sans-font-face` [Face]
Sans serif font face.

`unicode-tokens-fraktur-font-face` [Face]
Fraktur font face.

`unicode-tokens-script-font-face` [Face]
Script font face.

4.5 Moving between Unicode and tokens

If you want to share text between applications (e.g., email some text from an Isabelle theory file which heavily uses symbols), it is useful to convert to and from Unicode with cut-and-paste operations. The default buffer cut and paste functions will copy the underlying text, which contains the tokens (ASCII format). To copy and convert or paste then convert back, use these commands:

```
Tokens -> Copy as unicode
Tokens -> Paste from unicode
```

Both of these are necessarily approximate. The buffer presentation may use additional controls (for super/subscript layout or bold fonts, etc), which cannot be converted. Pasting relies on being able to identify a unique token mapped from a single Unicode character; the token table may not include such an entry, or may be ambiguous.

`unicode-tokens-copy` *beg end* [Command]
Copy presentation of region between *beg* and *end*.
This is an approximation; it makes assumptions about the behaviour of symbol compositions, and will lose layout information.

`unicode-tokens-paste` [Command]
Paste text from clipboard, converting Unicode to tokens where possible.

If you are using a mixture of "real" Unicode and tokens like this you may want to be careful to check the buffer contents: the command `unicode-tokens-highlight-unicode` helps you to manage this. It is available on the Tokens menu as

```
Tokens -> Highlight Real Unicode Chars
```

Alternative ways to check are to toggle the display of tokens using

```
Tokens -> Reveal Symbol Tokens
```

(the similar entry for `Control Tokens` displays tokens being used to control layout). Or simply toggle the tokens mode, which will leave the true Unicode tokens untouched.

`unicode-tokens-highlight-unicode` [Variable]
Non-nil to highlight Unicode characters.

4.6 Finding available tokens shortcuts and symbols

Two commands (both on the Tokens menu) allow you to see the tokens and shortcuts available:

```
Tokens -> List Tokens
Tokens -> List Shortcuts
```

Additionally, you can view the complete Unicode character set available in the default Emacs font, with

```
Tokens -> List Unicode Characters
```

(this uses a list adapted from Norman Walsh's `unichars.el`).

Note that the Unicode Tokens modes displays symbols defined by symbol tokens in a special font.

`unicode-tokens-list-tokens` [Command]
Show a buffer of all tokens.

`unicode-tokens-list-shortcuts` [Command]
Show a buffer of all the shortcuts available.

`unicode-tokens-list-unicode-chars` [Command]
Insert each Unicode character into a buffer.
Lets you see which characters are available for literal display in your emacs font.

(fn)

4.7 Selecting suitable fonts

The precise set of symbol glyphs that are available to you will depend in complicated ways on your operating system, Emacs version, configuration options used when Emacs was compiled, installed font sets, and (even) command line options used to start Emacs. So it is hard to give comprehensive and accurate advice in this manual. In general, things work *much* better with Emacs 23 than earlier versions.

To improve flexibility, Unicode Tokens mode allows you to select another font to display symbols from the default font that is used to display text in the buffer. This is the font that is configured by the menu

```
Tokens -> Set Fonts -> Symbol
```

its customization name is `unicode-tokens-symbol-font-face`, but notice that only the font family aspect of the face is used. Similarly, other fonts can be configured for controling different font families (script, fraktur, etc).

For symbols, good results are possible by using a proportional font for displaying symbols that has many symbol glyphs, for example the main font StixGeneral font from the Stix Fonts project (http://www.stixfonts.org/). At the time of writing you can obtain a beta version of these fonts in TTF format from http://olegueret.googlepages.com/stixfonts-ttf. On recent Linux distributions and with an Emacs 23 build that uses Xft, simply copy these `ttf` files into the `.fonts` directory inside your home directory to make them available.

Another font I like is **DejaVu Sans Mono**. It covers all of the standard Isabelle symbols. Some of the symbols are currently not perfect; however this font is an open source effort so users can contribute or suggest improvements. See http://dejavu-fonts.org.

If you are stuck with Emacs 22, you need to use the *fontset* mechanism which configures sets of fonts to use for display. The default font sets may not include enough symbols (typical symptom: symbols display as empty boxes). To address this, the menu command

```
Tokens -> Set Fonts -> Make Fontsets
```

constructs a number of fontsets at particular point sizes, based on several widely available fonts. See `pg-fontsets.el` for the code. After running this command, you can select from additional fontsets from the menus for changing fonts.

For further suggestions, please search (and contribute!) to the Proof General wiki at http://proofgeneral.inf.ed.ac.uk/wiki.

`unicode-tokens-symbol-font-face` [Face]
The default font used for symbols. Only :family and :slant attributes are used.

`unicode-tokens-font-family-alternatives` [Variable]
Not documented.

5 Support for other Packages

Proof General makes some configuration for other Emacs packages which provide various useful facilities that can make your editing more effective.

Sometimes this configuration is purely at the proof assistant specific level (and so not necessarily available), and sometimes it is made using Proof General settings.

When adding support for a new proof assistant, we suggest that these other packages are supported, as a convention.

The packages currently supported include `font-lock`, `imenu` and `speedbar`, `outline-mode`, `completion`, and `etags`.

5.1 Syntax highlighting

Proof script buffers are decorated (or *fontified*) with colours, bold and italic fonts, etc, according to the syntax of the proof language and the settings for `font-lock-keywords` made by the proof assistant specific portion of Proof General. Moreover, Proof General usually decorates the output from the proof assistant, also using `font-lock`.

To automatically switch on fontification in Emacs, you may need to engage `M-x global-font-lock-mode`.

By the way, the choice of colour, font, etc, for each kind of markup is fully customizable in Proof General. Each *face* (Emacs terminology) is controlled by its own customization setting. You can display a list of all of them using the customize menu:

```
Proof General -> Advanced -> Customize -> Faces -> Proof Faces.
```

5.2 Imenu and Speedbar

The Emacs package `imenu` (Index Menu) provides a menu built from the names of entities (e.g., theorems, definitions, etc) declared in a buffer. This allows easy navigation within the file. Proof General configures both packages automatically so that you can quickly jump to particular proofs in a script buffer.

(Developers note: the automatic configuration is done with the settings `proof-goal-with-hole-regexp` and `proof-save-with-hole-regexp`. Better configuration may be made manually with several other settings, see the *Adapting Proof General* manual for further details).

To use Imenu, select the option

```
Proof General -> Quick Options -> Minor Modes -> Index Menu
```

This adds an "Index" menu to the main menu bar for proof script buffers. You can also use *`M-x imenu`* for keyboard-driven completion of tags built from names in the buffer.

Speedbar displays a file tree in a separate window on the display, allowing quick navigation. Middle/double-clicking or pressing `+` on a file icon opens up to display tags (definitions, theorems, etc) within the file. Middle/double-clicking on a file or tag jumps to that file or tag.

To use Speedbar, use

```
Proof General -> Quick Options -> Minor Modes -> Speedbar
```

If you prefer the old fashioned way, 'M-x speedbar' does the same job.

For more information about Speedbar, see http://cedet.sourceforge.net/speedbar.shtml.

5.3 Support for outline mode

Proof General configures Emacs variables (`outline-regexp` and `outline-heading-end-regexp`) so that outline minor mode can be used on proof script files. The headings taken for outlining are the "goal" statements at the start of goal-save sequences, see Section 2.3.2 [Goal-save sequences], page 10. If you want to use `outline` to hide parts of the proof script in the *locked* region, you need to disable `proof-strict-read-only`.

Use *M-x outline-minor-mode* to turn on outline minor mode. Functions for navigating, hiding, and revealing the proof script are available in menus.

Please note that outline-mode may not work well in processed proof script files, because of read-only restrictions of the protected region. This is an inherent problem with outline because it works by modifying the buffer. If you want to use outline with processed scripts, you can turn off the `Strict Read Only` option.

See See Info file '`(emacs)`', node '`Outline Mode`' for more information about outline mode.

5.4 Support for completion

You might find the *completion* facility of Emacs useful when you're using Proof General. The key *C-RET* is defined to invoke the `complete` command. Pressing *C-RET* cycles through completions displaying hints in the minibuffer.

Completions are filled in according to what has been recently typed, from a database of symbols. The database is automatically saved at the end of a session.

Proof General has the additional facility for setting a completion table for each supported proof assistant, which gets loaded into the completion database automatically. Ideally the completion table would be set from the running process according to the identifiers available are within the particular context of a script file. But until this is available, this table may be set to contain a number of standard identifiers available for your proof assistant.

The setting *PA*`-completion-table` holds the list of identifiers for a proof assistant. The function `proof-add-completions` adds these into the completion database.

`PA-completion-table` [Variable]
List of identifiers to use for completion for this proof assistant.
Completion is activated with M-x complete.

If this table is empty or needs adjusting, please make changes using '`customize-variable`' and post suggestions at http://proofgeneral.inf.ed.ac.uk/trac

The completion facility uses a library '`completion.el`' which usually ships with Emacs, and supplies the `complete` function.

`complete` [Command]
Fill out a completion of the word before point.
Point is left at end. Consecutive calls rotate through all possibilities. Prefix args:

C-u	leave point at the beginning of the completion, not the end.
a number	rotate through the possible completions by that amount
0	same as -1 (insert previous completion)

See the comments at the top of '`completion.el`' for more info.

5.5 Support for tags

An Emacs "tags table" is a description of how a multi-file system is broken up into files. It lists the names of the component files and the names and positions of the functions (or other named subunits) in each file. Grouping the related files makes it possible to search or replace through all the files with one command. Recording the function names and positions makes possible the *M-.* command which finds the definition of a function by looking up which of the files it is in.

Some instantiations of Proof General (currently LEGO and Coq) are supplied with external programs ('`legotags`' and '`coqtags`') for making tags tables. For example, invoking '`coqtags *.v`' produces a file '`TAGS`' for all files '`*.v`' in the current directory. Invoking '`coqtags ‘find . -name \*.v‘`' produces a file '`TAGS`' for all files ending in '`.v`' in the current directory structure. Once a tag table has been made for your proof developments, you can use the Emacs tags mechanisms to find tags, and complete symbols from tags table.

One useful key-binding you might want to make is to set the usual tags completion key *M-tab* to run `tag-complete-symbol` to use completion from names in the tag table. To set this binding in Proof General script buffers, put this code in your '`.emacs`' file:

```
(add-hook 'proof-mode-hook
  (lambda () (local-set-key '(meta tab) 'tag-complete-symbol)))
```

Since this key-binding interferes with a default binding that users may already have customized (or may be taken by the window manager), Proof General doesn't do this automatically.

Apart from completion, there are several other operations on tags. One common one is replacing identifiers across all files using `tags-query-replace`. For more information on how to use tags, See Info file '`(emacs)`', node '`Tags`'.

To use tags for completion at the same time as the completion mechanism mentioned already, you can use the command *M-x add-completions-from-tags-table*.

`add-completions-from-tags-table` [Command]

Add completions from the current tags table.

6 Subterm Activation and Proof by Pointing

This chapter describes what you can do from inside the goals buffer, providing support for these features exists for your proof assistant.

As of Proof General 4.0, this support only exists for LEGO and proof-by-pointing functionality has been temporarily removed from the interface. If you would like to see subterm activation support for Proof General in another proof assistant, please petition the developers of that proof assistant to provide it!

6.1 Goals buffer commands

When you are developing a proof, the input focus (Emacs cursor) is usually on the script buffer. Therefore Proof General binds some mouse buttons for commands in the goals buffer, to avoid the need to move the cursor between buffers.

The mouse bindings are these:

mouse-1 `pg-goals-button-action`

C-mouse-3
`proof-undo-and-delete-last-successful-command`

C-S-mouse-1
`pg-identifier-under-mouse-query`

Where *mouse-1* indicates the left mouse button, and *mouse-3* indicates the right hand mouse button. The functions available provide a way to construct commands automatically (`pg-goals-button-action`) and to inspect identifiers (`pg-identifier-under-mouse-query`) as the Info toolbar button does.

Proof-by-pointing is a cute idea. It lets you automatically construct parts of a proof by clicking. You can ask the proof assistant to try to do a step in the proof, based on where you click. If you don't like the command which was inserted into the script, you can comment use the control key with the right button to undo the step and delete it from your script (`proof-undo-and-delete-last-successful-command`).

Proof-by-pointing may construct several commands in one go. These are sent back to the proof assistant altogether and appear as a single step in the proof script. However, if the proof is later replayed (without using PBP), the proof-by-pointing constructions will be considered as separate proof commands, as usual.

The main function for proof-by-pointing is `pg-goals-button-action`.

`pg-goals-button-action` *event* [Command]
Construct a proof-by-pointing command based on the mouse-click *event*.
This function should be bound to a mouse button in the Proof General goals buffer.

The *event* is used to find the smallest subterm around a point. A position code for the subterm is sent to the proof assistant, to ask it to construct an appropriate proof command. The command which is constructed will be inserted at the end of the locked region in the proof script buffer, and immediately sent back to the proof assistant. If it succeeds, the locked region will be extended to cover the proof-by-pointing command, just as for any proof command the user types by hand.

Proof-by-pointing uses markup describing the term structure of the concrete syntax output by the proof assistant. This markup is useful in itself: it allows you to explore the structure of a term using the mouse (the smallest subexpression that the mouse is over is highlighted), and easily copy subterms from the output to a proof script.

`pg-identifier-under-mouse-query` *event* [Command]
Query the prover about the identifier near mouse click *event*.

7 Customizing Proof General

There are two ways of customizing Proof General: it can be customized for a user's preferences using a particular proof assistant, or it can be customized by a developer to add support for a new proof assistant. The latter kind of customization we call instantiation, or *adapting*. See the *Adapting Proof General* manual for how to do this. Here we cover the user-level customization for Proof General.

There are two kinds of user-level settings in Proof General:

- Settings that apply *globally* to all proof assistants.
- those that can be adjusted for each proof assistant *individually*.

The first sort have names beginning with `proof-`. The second sort have names which begin with a symbol corresponding to the proof assistant: for example, `isa-`, `coq-`, etc. The symbol is the root of the mode name. See Section 1.2 [Quick start guide], page 3, for a table of the supported modes. To stand for an arbitrary proof assistant, we write *`PA-`* for these names.

In this chapter we only consider the generic settings: ones which apply to all proof assistants (globally or individually). The support for a particular proof assistant may provide extra individual customization settings not available in other proof assistants. See the chapters covering each assistant for details of those settings.

7.1 Basic options

Proof General has some common options which you can toggle directly from the menu:

```
Proof-General -> Quick Options
```

The effect of changing one of these options will be seen immediately (or in the next proof step). The window-control options on this menu are described shortly. See Section 7.3 [Display customization], page 38.

To save the current settings for these options (only), use the Save Options command in the submenu:

```
Proof General -> Quick Options -> Save Options
```

or `M-x customize-save-customized`.

The options on this sub-menu are also available in the complete user customization options group for Proof General. For this you need to know a little bit about how to customize in Emacs.

7.2 How to customize

Proof General uses the Emacs customization library to provide a friendly interface. You can access all the customization settings for Proof General via the menu:

```
Proof-General -> Advanced -> Customize
```

Using the customize facility is straightforward. You can select the setting to customize via the menus, or with `M-x customize-variable`. When you have selected a setting, you are shown a buffer with its current value, and facility to edit it. Once you have edited it, you can use the special buttons *set*, *save* and *done*. You must use one of *set* or *save* to get any effect. The *save* button stores the setting in your '`.emacs`' file. The command *`M-x customize-save-customized`* or Emacs menubar item `Options -> Save Options` saves all settings you have edited.

A technical note. In the customize menus, the variable names mentioned later in this chapter may be abbreviated — the `"proof-"` or similar prefixes are omitted. Also, some of the option settings may have more descriptive names (for example, *on* and *off*) than the low-level lisp values (non-`nil`, `nil`) which are mentioned in this chapter. These features make customize rather more friendly than raw lisp.

You can also access the customize settings for Proof General from other (non-script) buffers. Use the menu:

```
Options -> Customize Emacs -> Top-level Customization Group
```

and select the `External` and then `Proof-General` groups.

The complete set of customization settings will only be available after Proof General has been fully loaded. Proof General is fully loaded when you visit a script file for the first time, or if you type *M-x load-library RET proof RET*.

For more help with customize, see See Info file '`(emacs)`', node '`Customization`'.

7.3 Display customization

By default, Proof General displays two buffers during scripting, in a split window on the display. One buffer is the script buffer. The other buffer is either the goals buffer (`*goals*`) or the response buffer (`*response*`). Proof General raises and switches between these last two automatically.

Proof General allows several ways to customize this default display model, by splitting the Emacs frames in different ways and maximising the amount of information shown, or by using multiple frames. The customization options are explained below; they are also available on the menu:

```
Proof-General -> Quick Options -> Display
```

and you can save your preferred default.

If your screen is large enough, you may prefer to display all three of the interaction buffers at once. This is useful, for example, to see output from the `proof-find-theorems` command at the same time as the subgoal list. Set the user option `proof-three-window-enable` to make Proof General keep both the goals and response buffer displayed.

If you prefer to switch windows and buffers manually when you want to see the prover output, you can customize the user option `proof-auto-raise-buffers` to prevent the automatic behaviour. You can browse interaction output by hovering the mouse over the command regions in the proof script.

`proof-auto-raise-buffers` [User Option]

If non-nil, automatically raise buffers to display latest output.
If this is not set, buffers and windows will not be managed by Proof General.

The default value is `t`.

`proof-three-window-enable` [User Option]

Whether response and goals buffers have dedicated windows.
If non-nil, Emacs windows displaying messages from the prover will not be switchable to display other windows.

This option can help manage your display.

Setting this option triggers a three-buffer mode of interaction where the goals buffer and response buffer are both displayed, rather than the two-buffer mode where they are switched between. It also prevents Emacs automatically resizing windows between proof steps.

If you use several frames (the same Emacs in several windows on the screen), you can force a frame to stick to showing the goals or response buffer.

The default value is `nil`.

Sometimes during script management, there is no response from the proof assistant to some command. In this case you might like the empty response window to be hidden so you have more room to see the proof script. The setting `proof-delete-empty-windows` helps you do this.

`proof-delete-empty-windows` [User Option]

If non-nil, automatically remove windows when they are cleaned.
For example, at the end of a proof the goals buffer window will be cleared; if this flag is set it will automatically be removed. If you want to fix the sizes of your windows you may want to set this variable to `'nil'` to avoid windows being deleted automatically. If you use multiple frames, only the windows in the currently selected frame will be automatically deleted.

The default value is `nil`.

This option only has an effect when you have set `proof-three-window-mode`.

If you are working on a machine with a window system, you can use Emacs to manage several *frames* on the display, to keep the goals buffer displayed in a fixed place on your screen and in a certain font, for example. A convenient way to do this is via the user option

`proof-multiple-frames-enable` [User Option]

Whether response and goals buffers have separate frames.
If non-nil, Emacs will make separate frames (screen windows) for the goals and response buffers, by altering the Emacs variable '`special-display-regexps`'.

The default value is `nil`.

Multiple frames work best when `proof-delete-empty-windows` is off and `proof-three-window-mode` is on.

Finally, there are two commands available which help to switch between buffers or refresh the window layout. These are on the menu:

```
Proof-General -> Buffers
```

`proof-display-some-buffers` [Command]

Display the reponse, trace, goals, or shell buffer, rotating.
A fixed number of repetitions of this command switches back to the same buffer. Also move point to the end of the response buffer if it's selected. If in three window or multiple frame mode, display two buffers. The idea of this function is to change the window->buffer mapping without adjusting window layout.

`proof-layout-windows` **&optional** *nohorizontalsplit* [Command]

Refresh the display of windows according to current display mode.

For single frame mode, this uses a canonical layout made by splitting Emacs windows vertically in equal proportions. You can then adjust the proportions by dragging the separating bars. In three pane mode, the canonical layout is to split both horizontally and vertically, to display the prover responses in two panes on the right-hand side, and the proof script in a taller pane on the left. A prefix argument will prevent the horizontal split, and result in three windows spanning the full width of the Emacs frame.

For multiple frame mode, this function obeys the setting of '`pg-response-eagerly-raise`', which see.

`proof-shrink-windows-tofit` [User Option]

If non-nil, automatically shrink output windows to fit contents.
In single-frame mode, this option will reduce the size of the goals and response windows to fit their contents.

The default value is `nil`.

`proof-colour-locked` [User Option]

If non-nil, colour the locked region with '`proof-locked-face`'.
If this is not set, buffers will have no special face set on locked regions.

The default value is `t`.

7.4 User options

Here is the complete set of user options for Proof General, apart from the three display options mentioned above.

User options can be set via the customization system already mentioned, via the old-fashioned `M-x edit-options` mechanism, or simply by adding `setq`'s to your '`.emacs`' file. The first approach is strongly recommended.

Unless mentioned, all of these settings can be changed dynamically, without needing to restart Emacs to see the effect. But you must use customize to be sure that Proof General reconfigures itself properly.

`proof-splash-enable` [User Option]
If non-nil, display a splash screen when Proof General is loaded.

The default value is `t`.

`proof-electric-terminator-enable` [User Option]
If non-nil, use electric terminator mode.
If electric terminator mode is enabled, pressing a terminator will automatically issue '`proof-assert-next-command`' for convenience, to send the command straight to the proof process. If the command you want to send already has a terminator character, you don't need to delete the terminator character first. Just press the terminator somewhere nearby. Electric!

The default value is `nil`.

`proof-toolbar-enable` [User Option]
If non-nil, display Proof General toolbar for script buffers.

The default value is `t`.

`proof-strict-read-only` [User Option]
Whether Proof General is strict about the read-only region in buffers.
If non-nil, an error is given when an attempt is made to edit the read-only region, except for the special value `'retract` which means undo first. If nil, Proof General is more relaxed (but may give you a reprimand!).

The default value is `retract`.

`proof-query-file-save-when-activating-scripting` [User Option]
If non-nil, query user to save files when activating scripting.

Often, activating scripting or executing the first scripting command of a proof script will cause the proof assistant to load some files needed by the current proof script. If this option is non-nil, the user will be prompted to save some unsaved buffers in case any of them corresponds to a file which may be loaded by the proof assistant.

You can turn this option off if the save queries are annoying, but be warned that with some proof assistants this may risk processing files which are out of date with respect to the loaded buffers!

The default value is `t`.

`PA-script-indent` [User Option]
If non-nil, enable indentation code for proof scripts.

The default value is `t`.

`PA-one-command-per-line` [User Option]
If non-nil, format for newlines after each command in a script.

The default value is `t`.

`proof-prog-name-ask` [User Option]
If non-nil, query user which program to run for the inferior process.

The default value is `nil`.

`proof-prog-name-guess` [User Option]
If non-nil, use '`proof-guess-command-line`' to guess '`proof-prog-name`'.
This option is compatible with '`proof-prog-name-ask`'. No effect if '`proof-guess-command-line`' is nil.

The default value is `nil`.

`proof-tidy-response` [User Option]
Non-nil indicates that the response buffer should be cleared often.
The response buffer can be set either to accumulate output, or to clear frequently.

With this variable non-nil, the response buffer is kept tidy by clearing it often, typically between successive commands (just like the goals buffer).

Otherwise the response buffer will accumulate output from the prover.

The default value is `t`.

`proof-keep-response-history` [User Option]
Whether to keep a browsable history of responses.
With this feature enabled, the buffers used for prover responses will have a history that can be browsed without processing/undoing in the prover. (Changes to this variable take effect after restarting the prover).

The default value is `nil`.

`pg-input-ring-size` [User Option]
Size of history ring of previous successfully processed commands.

The default value is `32`.

`proof-general-debug` [User Option]
Non-nil to run Proof General in debug mode.
This changes some behaviour (e.g. markup stripping) and displays debugging messages in the response buffer. To avoid erasing messages shortly after they're printed, set '`proof-tidy-response`' to nil. This is only useful for PG developers.

The default value is `nil`.

`proof-follow-mode` [User Option]
Choice of how point moves with script processing commands.
One of the symbols: `'locked`, `'follow`, `'followdown`, `'ignore`.

If `'locked`, point sticks to the end of the locked region. If `'follow`, point moves just when needed to display the locked region end. If `'followdown`, point if necessary to stay in writeable region If `'ignore`, point is never moved after movement commands or on errors.

If you choose `'ignore`, you can find the end of the locked using M-x `proof-goto-end-of-locked`

The default value is `locked`.

`proof-auto-action-when-deactivating-scripting` [User Option]
If `'retract` or `'process`, do that when deactivating scripting.

With this option set to `'retract` or `'process`, when scripting is turned off in a partly processed buffer, the buffer will be retracted or processed automatically.

With this option unset (nil), the user is questioned instead.

Proof General insists that only one script buffer can be partly processed: all others have to be completely processed or completely unprocessed. This is to make sure that handling of multiple files makes sense within the proof assistant.

NB: A buffer is completely processed when all non-whitespace is locked (coloured blue); a buffer is completely unprocessed when there is no locked region.

The default value is `nil`.

`proof-rsh-command` [User Option]

Shell command prefix to run a command on a remote host.
For example,

```
ssh bigjobs
```

Would cause Proof General to issue the command '`ssh bigjobs isabelle`' to start Isabelle remotely on our large compute server called '`bigjobs`'.

The protocol used should be configured so that no user interaction (passwords, or whatever) is required to get going. For proper behaviour with interrupts, the program should also communicate signals to the remote host.

The default value is `nil`.

7.5 Changing faces

The numerous fonts and colours that Proof General uses are configurable. If you alter faces through the customize menus (or the command *`M-x customize-face`*), only the particular kind of display in use (colour window system, monochrome window system, console, ...) will be affected. This means you can keep separate default settings for each different display environment where you use Proof General.

As well as the faces listed below, Proof General may use the regular `font-lock-` faces (eg `font-lock-keyword-face`, `font-lock-variable-name-face`, etc) for fontifying the proof script or proof assistant output. These can be altered to your taste just as easily, but note that changes will affect all other modes which use them!

7.5.1 Script buffer faces

`proof-queue-face` [Face]

Face for commands in proof script waiting to be processed.

`proof-locked-face` [Face]

Face for locked region of proof script (processed commands).

`proof-script-sticky-error-face` [Face]

Proof General face for marking an error in the proof script.

`proof-script-highlight-error-face` [Face]

Proof General face for highlighting an error in the proof script.

`proof-mouse-highlight-face` [Face]

General mouse highlighting face used in script buffer.

`proof-highlight-dependent-face` [Face]

Face for showing (backwards) dependent parts.

`proof-highlight-dependency-face` [Face]

Face for showing (forwards) dependencies.

`proof-declaration-name-face` [Face]
Face for declaration names in proof scripts.
Exactly what uses this face depends on the proof assistant.

`proof-tacticals-name-face` [Face]
Face for names of tacticals in proof scripts.
Exactly what uses this face depends on the proof assistant.

7.5.2 Goals and response faces

`proof-error-face` [Face]
Face for error messages from proof assistant.

`proof-warning-face` [Face]
Face for warning messages.
Warning messages can come from proof assistant or from Proof General itself.

`proof-debug-message-face` [Face]
Face for debugging messages from Proof General.

`proof-boring-face` [Face]
Face for boring text in proof assistant output.

`proof-active-area-face` [Face]
Face for showing active areas (clickable regions), outside of subterm markup.

`proof-eager-annotation-face` [Face]
Face for important messages from proof assistant.

The slightly bizarre name of the last face comes from the idea that while large amounts of output are being sent from the prover, some messages should be displayed to the user while the bulk of the output is hidden. The messages which are displayed may have a special annotation to help Proof General recognize them, and this is an "eager" annotation in the sense that it should be processed as soon as it is observed by Proof General.

7.6 Tweaking configuration settings

This section is a note for advanced users.

Configuration settings are the per-prover customizations of Proof General. These are not intended to be adjusted by the user. But occasionally you may like to test changes to these settings to improve the way Proof General works. You may want to do this when a proof assistant has a flexible proof script language in which one can define new tactics or even operations, and you want Proof General to recognize some of these which the default settings don't mention. So please feel free to try adjusting the configuration settings and report to us if you find better default values than the ones we have provided.

The configuration settings appear in the customization group `prover-config`, or via the menu

```
Proof-General -> Internals ->  Prover Config
```

One basic example of a setting you may like to tweak is:

`proof-assistant-home-page` [Variable]
Web address for information on proof assistant.
Used for Proof General's help menu.

Most of the others are more complicated. For more details of the settings, see *Adapting Proof General* for full details. To browse the settings, you can look through the customization groups `prover-config`, `proof-script` and `proof-shell`. The group `proof-script` contains the configuration variables for scripting, and the group `proof-shell` contains those for interacting with the proof assistant.

Unfortunately, although you can use the customization mechanism to set and save these variables, saving them may have no practical effect because the default settings are mostly hardwired into the proof assistant code. Ones we expect may need changing appear as proof assistant specific configurations. For example, `proof-assistant-home-page` is set in the LEGO code from the value of the customization setting `lego-www-home-page`. At present there is no easy way to save changes to other configuration variables across sessions, other than by editing the source code. (In future versions of Proof General, we plan to make all configuration settings editable in Customize, by shadowing the settings as prover specific ones using the *`PA-`* mechanism).

8 Hints and Tips

Apart from the packages officially supported in Proof General, many other features of Emacs are useful when using Proof General, even though they need no specific configuration for Proof General. It is worth taking a bit of time to explore the Emacs manual to find out about them.

Here we provide some hints and tips for a couple of Emacs features which users have found valuable with Proof General. Further contributions to this chapter are welcomed!

8.1 Adding your own keybindings

Proof General follows Emacs convention for file modes in using C-C prefix key-bindings for its own functions, which is why some of the default keyboard short-cuts are quite lengthy.

Some users may prefer to add additional key-bindings for shorter sequences. This can be done interactively with the command `M-x local-set-key`, or for longevity, by adding code like this to your '`.emacs`' file:

```
(eval-after-load "proof-script" '(progn
 (define-key proof-mode-map [(control n)]
                            'proof-assert-next-command-interactive)
 (define-key proof-mode-map [(control b)]
                            'proof-undo-last-successful-command)
 ))
```

This lisp fragment adds bindings for every buffer in proof script mode (the Emacs keymap is called `proof-mode-map`). To just affect one prover, use a keymap name like `isar-mode-map` and evaluate after the library `isar` has been loaded.

To find the names of the functions you may want to bind, look in this manual, or query current bindings interactively with *`C-h k`*. This command (`describe-key`) works for menu operations as well; also use it to discover the current key-bindings which you're losing by declarations such as those above. By default, *`C-n`* is `next-line` and *`C-b`* is `backward-char-command`; neither are really needed if you have working cursor keys.

If your keyboard has a *super* modifier (on my PC keyboard it has a Windows symbol and is next to the control key), you can freely bind keys on that modifier globally (since none are used by default). Use lisp like this:

```
(global-set-key [?\s-l] 'maths-menu-insert-lambda)
(global-set-key [?\s-l] 'maths-menu-insert-lambda)

(global-set-key [?\s-l] 'maths-menu-insert-lambda)
(global-set-key [?\s-L] 'maths-menu-insert-Lambda)
(global-set-key [?\s-D] 'maths-menu-insert-Delta)

(global-set-key [?\s-a] 'maths-menu-insert-for-all)
(global-set-key [?\s-e] 'maths-menu-insert-there-exists)
(global-set-key [?\s-t] 'maths-menu-insert-down-tack)
(global-set-key [?\s-b] 'maths-menu-insert-up-tack)

(global-set-key [?\s-\#] 'maths-menu-insert-music-sharp-sign)
(global-set-key [?\s-\.] 'maths-menu-insert-horizontal-ellipsis)

(global-set-key [?\s-3] 'proof-three-window-toggle)
```

This defines a bunch of short-cuts for inserting symbols taken from the Maths Menu, see Chapter 4 [Unicode symbols and special layout support], page 25 and a short-cut for enabling three window mode, see Section 7.3 [Display customization], page 38.

8.2 Using file variables

A very convenient way to customize file-specific variables is to use the File Variables (See Info file '`(emacs)`', node '`File Variables`'). This feature of Emacs allows to specify the values to use for certain Emacs variables when a file is loaded. Those values are written as a list at the end of the file.

For example, in projects involving multiple directories, it is often useful to set the variables `proof-prog-name`, `proof-prog-args` and `compile-command` for each file. Here is an example for Coq users: for the file '`.../dir/bar/foo.v`', if you want Coq to be started with the path `.../dir/theories/` added in the libraries path (`"-I"` option), you can put at the end of '`foo.v`':

```
(*
*** Local Variables: ***
*** coq-prog-name: "coqtop" ***
*** coq-prog-args: ("-emacs" "-I" "../theories") ***
*** End: ***
*)
```

That way the good command is called when the scripting starts in '`foo.v`'. Notice that the command argument `"-I" "../theories"` is specific to the file '`foo.v`', and thus if you set it via the configuration tool, you will need to do it each time you load this file. On the contrary with this method, Emacs will do this operation automatically when loading this file. Please notice that all the strings above should never contain spaces see documentation of variables `proof-prog-name` and `proof-prog-args`.

Extending the previous example, if the makefile for '`foo.v`' is located in directory '`.../dir/`', you can add the right compile command. And if you want a non standard coq executable to be used, here is what you should put in variables:

```
(*
 Local Variables:
 coq-prog-name: "../../coqsrc/bin/coqtop"
 coq-prog-args: "-emacs" "-I" "../theories"
 compile-command: "make -C .. -k bar/foo.vo"
 End:
*)
```

And then the right call to make will be done if you use the *`M-x compile`* command. Notice that the lines are commented in order to be ignored by the proof assistant. It is possible to use this mechanism for any other buffer local variable. See Info file '`emacs`', node '`File Variables`'.

8.3 Using abbreviations

A very useful package of Emacs supports automatic expansions of abbreviations as you type, See Info file '`(emacs)`', node '`Abbrevs`'.

For example, the proof assistant Coq has many command strings that are long, such as "reflexivity," "Inductive," "Definition" and "Discriminate." Here is a part of the Coq Proof General abbreviations:

```
"abs" "absurd "
"ap" "apply "
"as" "assumption"
```

The above list was taken from the file that Emacs saves between sessions. The easiest way to configure abbreviations is as you write, by using the key presses *C-x a g* (`add-global-abbrev`) or *C-x a i g* (`inverse-add-global-abbrev`). To enable automatic expansion of abbreviations (which can be annoying), the `Abbrev` minor mode, type *M-x abbrev-mode RET*. When you are not in Abbrev mode you can expand an abbreviation by pressing *C-x '* (`expand-abbrev`). See the Emacs manual for more details.

Coq Proof General has a special experimental feature called "Holes" which makes use of the abbreviation mechanism and includes a large list of command abbreviations. See Section 10.4 [Holes feature], page 52, for details. With other provers, you may use the standard Emacs commands above to set up your own abbreviation tables.

9 LEGO Proof General

LEGO proof script mode is a mode derived from proof script mode for editing LEGO scripts. An important convention is that proof script buffers *must* start with a module declaration. If the proof script buffer's file name is '`fermat.l`', then it must commence with a declaration of the form

```
Module fermat;
```

If, in the development of the module '`fermat`', you require material from other module e.g., '`lib_nat`' and '`galois`', you need to specify this dependency as part of the module declaration:

```
Module fermat Import lib_nat galois;
```

No need to worry too much about efficiency. When you retract back to a module declaration to add a new import item, LEGO does not actually retract the previously imported modules. Therefore, reasserting the extended module declaration really only processes the newly imported modules.

Using the LEGO Proof General, you never ever need to use administrative LEGO commands such as '`Forget`', '`ForgetMark`', '`KillRef`', '`Load`', '`Make`', '`Reload`' and '`Undo`' again[1].

9.1 LEGO specific commands

In addition to the commands provided by the generic Proof General (as discussed in the previous sections) the LEGO Proof General provides a few extensions. In proof scripts, there are some abbreviations for common commands:

C-c C-a C-i
: intros

C-c C-a C-I
: Intros

C-c C-a C-R
: Refine

9.2 LEGO tags

You might want to ask your local system administrator to tag the directories '`lib_Prop`', '`lib_Type`' and '`lib_TYPE`' of the LEGO library. See Section 5.5 [Support for tags], page 33, for further details on tags.

9.3 LEGO customizations

We refer to chapter Chapter 7 [Customizing Proof General], page 37, for an introduction to the customisation mechanism. In addition to customizations at the generic level, for LEGO you can also customize:

`lego-tags` [User Option]
: The directory of the *tags* table for the *lego* library

The default value is `"/usr/lib/lego/lib_Type/"`.

`lego-www-home-page` [Variable]
: Lego home page URL.

[1] And please, don't even think of including those in your LEGO proof script!

10 Coq Proof General

Coq Proof General is an instantiation of Proof General for the Coq proof assistant. It supports most of the generic features of Proof General.

10.1 Coq-specific commands

Coq Proof General supplies the following key-bindings:

`C-c C-a C-i`
: Inserts "Intros "

`C-c C-a C-a`
: Inserts "Apply "

`C-c C-a C-s`
: Inserts "Section "

`C-c C-a C-e`
: Inserts "End <section-name>." (this should work well with nested sections).

`C-c C-a C-o`
: Prompts for a SearchIsos argument.

10.2 Editing multiple proofs

Coq allows the user to enter top-level commands while editing a proof script. For example, if the user realizes that the current proof will fail without an additional axiom, he or she can add that axiom to the system while in the middle of the proof. Similarly, the user can nest lemmas, beginning a new lemma while in the middle of an earlier one, and as the lemmas are proved or their proofs aborted they are popped off a stack.

Coq Proof General supports this feature of Coq. Top-level commands entered while in a proof are well backtracked. If new lemmas are started, Coq Proof General lets the user work on the proof of the new lemma, and when the lemma is finished it falls back to the previous one. This is supported to any nesting depth that Coq allows.

Warning! Using Coq commands for navigating inside the different proofs (`Resume` and especially `Suspend`) are not supported, backtracking will break syncronization.

Special note: The old feature that moved nested proofs outside the current proof is disabled.

10.3 User-loaded tactics

Another feature that Coq allows is the extension of the grammar of the proof assistant by new tactic commands. This feature interacts with the proof script management of Proof General, because Proof General needs to know when a tactic is called that alters the proof state. When the user tries to retract across an extended tactic in a script, the algorithm for calculating how far to undo has a default behavior that is not always accurate in proof mode: do `"Undo"`.

Coq Proof General does not currently support dynamic tactic extension in Coq: this is desirable but requires assistance from the Coq core. Instead we provide a way to add tactic and command names in the '`.emacs`' file. Four Configurable variables allows to register personal new tactics and commands into four categories:

- *state changing commands*, which need `"Back"` to be backtracked;
- *state changing tactics*, which need `"Undo"` to be backtracked;
- *state preserving commands*, which do not need to be backtracked;
- *state preserving tactics*, which do not need to be backtracked;

We give an example of existing commands that fit each category.

- `coq-user-state-preserving-commands`: example: `"Print"`
- `coq-user-state-changing-commands`: example: `"Require"`
- `coq-user-state-changing-tactics`: example: `"Intro"`
- `coq-user-state-preserving-tactics`: example: `"Idtac"`

This variables are regexp string lists. See their documentations in emacs (`C-h v coq-user...`) for details on how to set them in your '`.emacs`' file.

Here is a simple example:

```
(setq coq-user-state-changing-commands
      '("MyHint" "MyRequire"))
(setq coq-user-state-preserving-commands
      '("Show\\s-+Mydata"))
```

The regexp character sequence `\\s-+` means "one or more whitespaces". See the Emacs documentation of `regexp-quote` for the syntax and semantics. WARNING: you need to restart Emacs to make the changes to these variables effective.

In case of losing synchronization, the user can use *C-c C-z* to move the locked region to the proper position, (`proof-frob-locked-end`, see Section 3.9 [Escaping script management], page 22) or *C-c C-v* to re-issue an erroneously back-tracked tactic without recording it in the script.

10.4 Holes feature

Holes are an experimental feature for complex expression editing by filling in templates. It is inspired from other tools, like Pcoq (`http://www-sop.inria.fr/lemme/pcoq/index.html`). The principle is simple, holes are pieces of text that can be "filled" by various means. The Coq command insertion menu system makes use of the holes system. Almost all holes operations are available in the Holes menu.

Notes: Holes make use of the Emacs abbreviation mechanism, it will work without problem if you don't have an abbrev table defined for Coq in your config files. Use `C-h v abbrev-file-name` to see the name of the abbreviation file.

If you already have such a table it won't be automatically overwritten (so that you keep your own abbreviations). But you must read the abbrev file given in the Proof General sources to be able to use the command insertion menus. You can do the following to merge your abbreviations with ProofGeneral's abbreviations: `M-x read-abbrev-file`, then select the file named `coq-abbrev.el` in the `ProofGeneral/coq` directory. At Emacs exit you will be asked if you want to save abbrevs; answer yes.

11 Isabelle Proof General

Isabelle Proof General supports major generic features of Proof General, including integration with Isabelle's theory loader for proper automatic multiple file handling.

Isabelle provides its own way to invoke Proof General via the `isabelle` command. Running `isabelle emacs` starts an Emacs session with Isabelle Proof General. The defaults may be changed by editing the Isabelle settings, see the Isabelle documentation for details.

Proof General for Isabelle manages Isar '`.thy`' files. Proof General provides reliable control over incremental execution of the text.

11.1 Choosing logic and starting isabelle

When you load an Isabelle theory file into Proof General, you may be prompted for the path to the program `isabelle` if it is not on the system `PATH` already. This is used to generate further information for invoking Isabelle, in particular, the list of available logics.

The Isabelle menu offers an easy way to select the invoked object logic. If you look at the menu:

```
Isabelle -> Logics ->
```

you should see the list of logics available to Isabelle. This menu is generated from the output of the command `isabelle findlogics`. (Similarly, the documentation menu is partly generated from `isabelle doc`). Instead of the menu, you can use the keyboard command `isabelle-chose-logic` to choose from the list.

The logics list is refreshed dynamically so you can select any newly built heap images in the same Emacs session. However, notice that the choices are greyed out while Isabelle is actually runnning — you can only switch to a new logic if you first exit Isabelle (similarly to Proof General, Isabelle operates with only one logic at a time).

Another way to set the logic before Isabelle is launched is using an Emacs local variable setting inside a comment at the top of the file, see the documentation of `isabelle-chosen-logic` below.

In case you do not have the `isabelle` program available or want to override its behaviour, you may set the variable `isabelle-program-name-override` to define the name of the executable used to start Isabelle. The standard options are and logic name are still appended.

`isabelle-program-name-override` [User Option]

Name of executable program to run Isabelle.

You can set customize this in case the automatic settings mechanism does not work for you, perhaps because isabelle is not on your path, or you are running it remotely.

The logic image name is tagged onto the end.

The default value is `nil`.

`isabelle-chosen-logic` [User Option]

Choice of logic to use with Isabelle.
If non-nil, added onto the Isabelle command line for invoking Isabelle.

You can set this as a file local variable, using a special comment at the top of your theory file, like this:

```
(* -*- isabelle-chosen-logic: "ZF" -*- *)
```

The default value is `nil`.

`isabelle-choose-logic` *logic* [Command]

Adjust isabelle-prog-name and `proof-prog-name` for running *logic*.

11.2 Isabelle commands

The Isabelle instance of Proof General supplies several specific help key bindings; these functions are offered within the prover help menu as well.

C-c C-a r Invokes Isar command `refute` on the current subgoal. Only available in HOL and derived logics.

C-c C-a C-q
Invokes Isar command `quickcheck` on the current subgoal.

C-c C-a C-d
Displays a draft document of the current theory.

C-c C-a C-p
Prints a draft document of the current theory.

C-c C-a h A
Shows available antiquotation commands and options.

C-c C-a h C
Shows the current Classical Reasoner context.

C-c C-a h I
Shows the current set of induct/cases rules.

C-c C-a h S
Shows the current Simplifier context.

C-c C-a h T
Shows the current set of transitivity rules (for calculational reasoning).

C-c C-a h a
Shows attributes available in current theory context.

C-c C-a h b
Shows all local term bindings.

C-c C-a h c
Shows all named local contexts (cases).

C-c C-a h f
Shows all local facts.

C-c C-a h i
Shows inner syntax of the current theory context (for types and terms).

C-c C-a h m
Shows proof methods available in current theory context.

C-c C-a h o
Shows all available commands of Isabelle's outer syntax.

C-c C-a h t
Shows theorems stored in the current theory node.

C-c C-a C-s
Invoke sledgehammer on first subgoal.

C-c C-a C-m
Find theorems containing given arguments (prompt in minibuffer). Invokes the `thms_containing` command. Arguments are separated by white space as usual in Isar.

`C-c C-a C-f`
: Find theorems containing (argument in form)

`C-c C-f` Find theorems: either of the above.

The following shortcuts insert control sequences into the text, modifying the appearance of individual symbols (single letters, mathematical entities etc.); the Tokens package will provide immediate visual feedback.

`C-c C-a C-b`
: Inserts `"\<^bold>"` (bold character)

`C-c C-a C-c`
: Inserts `"\<^loc>"` ("local" control)

`C-c C-a C-u`
: Inserts `"\<^sup>"` (superscript character)

`C-c C-a C-l`
: Inserts `"\<^sub>"` (subscript character)

`C-c C-a u` Inserts `"\<^bsup> \<^esup>"` (superscript string)

`C-c C-a l` Inserts `"\<^bsub> \<^esub>"` (subscript string)

`C-c C-a C-i`
: Inserts `"\<^isub>"` (identifier subscript letter)

`C-c C-a C-r`
: Inserts `"\<^raw:>"` (raw LaTeX text)

`C-c C-a C-a`
: Inserts `"@{text ""}"` (anti-quotation).

`C-c C-a C-x`
: Inserts `"ML {* *}"` (inline ML code).

Command termination via '`;`' is an optional feature of Isar syntax. Neither Isabelle nor Proof General require semicolons to do their job. The following command allows to get rid of command terminators in existing texts.

`isar-strip-terminators` [Command]
Remove explicit Isabelle/Isar command terminators '`;`' from the buffer.

11.3 Isabelle settings

The Isabelle menu also contains a `Settings` submenu, which allows you to configure things such as the behaviour of Isabelle's term pretty printer (display of types, sorts, etc). Note that you won't see this sub-menu until Isabelle has been started, because it is generated by Isabelle itself. Proof General, on the other hand, is responsible for recording any settings that are configured when you select `Isabelle -> Settings -> Save Settings`. They are stored along with the other Emacs customization settings.

11.4 Isabelle customizations

Here are some of the other user options specific to Isabelle. You can set these as usual with the customization mechanism.

`isabelle-web-page` [Variable]
URL of web page for Isabelle.

12 HOL Proof General

HOL Proof General is a "technology demonstration" of Proof General for HOL4 (aka HOL98). This means that only a basic instantiation has been provided, and that it is not yet supported as a maintained instantiation of Proof General.

HOL Proof General has basic script management support, with a little bit of decoration of scripts and output. It does not rely on a modified version of HOL, so the pattern matching may be fragile in certain cases. Support for multiple files deduces dependencies automatically, so there is no interaction with the HOL make system yet.

See the '`example.sml`' file for a demonstration proof script which works with Proof General.

Note that HOL proof scripts often use batch-oriented single step tactic proofs, but Proof General does not (yet) offer an easy way to edit these kind of proofs. They will replay simply as a single step proof and you will need to convert from the interactive to batch form as usual if you wish to obtain batch proofs. Also note that Proof General does not contain an SML parser, so there can be problems if you write complex ML in proof scripts.

HOL Proof General may work with variants of HOL other than HOL98, but is untested. Probably a few of the settings would need to be changed in a simple way, to cope with small differences in output between the systems. (Please let us know if you modify the HOL98 version for another variant of HOL).

Perhaps somebody from the HOL community is willing to adopt HOL Proof General and support and improve it. Please volunteer!

13 Shell Proof General

This instance of Proof General is not really for proof assistants at all, but simply provided as a handy way to use a degenerate form of script management with other tools.

Suppose you have a software tool of some kind with a command line interface, and you want to demonstrate several example uses of it, perhaps at a conference. But the command lines for your tool may be quite complicated, so you do not want to type them in live. Instead, you just want to cut and paste from a pre-recorded list. But watching somebody cut and paste commands into a window is almost as tedious as watching them type those commands!

Shell Proof General comes to the rescue. Simply record your commands in a file with the extension `.pgsh`, and load up Proof General. Now use the toolbar to send each line of the file to your tool, and have the output displayed clearly in another window. Much easier and more pleasant for your audience to watch!

If you wish, you may adjust the value of `proof-prog-name` in '`pgshell.el`' to launch your program rather than the shell interpreter.

We welcome feedback and suggestions concerning this subsidiary provision in Proof General. Please recommend it to your colleagues (e.g., the model checking crew).

Appendix A Obtaining and Installing

Proof General has its own home page hosted at Edinburgh. Visit this page for the latest news!

A.1 Obtaining Proof General

You can obtain Proof General from the URL

`http://proofgeneral.inf.ed.ac.uk`.

The distribution is available in as a tarball. It may be redistributed by third party packagers in other forms.

Two versions are available:

- The current stable version,
  `http://proofgeneral.inf.ed.ac.uk/ProofGeneral-stable.tgz`
- The latest development version,
  `http://proofgeneral.inf.ed.ac.uk/ProofGeneral-latest.tgz`

The source tarball includes the generic elisp code, and code for LEGO, Coq, Isabelle, and other provers. Also included are installation instructions (reproduced in brief below) and this documentation.

For access to the source code repository, please see the Proof General web page `http://proofgeneral.inf.ed.ac.uk/devel`.

A.2 Installing Proof General from tarball

Copy the distribution to some directory *mydir*. Unpack it there. For example:

```
# cd mydir
# tar -xpzf ProofGeneral-version.tgz
```

If you downloaded the version called *latest*, you'll find it unpacks to a numeric version number.

Proof General will now be in some subdirectory of *mydir*. The name of the subdirectory will depend on the version number of Proof General. For example, it might be '`ProofGeneral-4.0`'. It's convenient to link it to a fixed name:

```
# ln -sf ProofGeneral-4.0 ProofGeneral
```

Now put this line in your '`.emacs`' file:

```
(load-file "mydir/ProofGeneral/generic/proof-site.el")
```

A.3 Setting the names of binaries

The `load-file` command you have added will load '`proof-site`' which sets the Emacs load path for Proof General and add auto-loads and modes for the supported assistants.

The default names for proof assistant binaries may work on your system. If not, you will need to set the appropriate variables. The easiest way to do this (and most other customization of Proof General) is via the Customize mechanism, see the menu item:

```
Proof-General -> Advanced -> Customize -> Name of Assistant -> Prog Name
```

The Proof-General menu is available from script buffers after Proof General is loaded. To load it manually, type

```
M-x load-library RET proof RET
```

If you do not want to use customize, simply add a line like this:

```
(setq coq-prog-name "/usr/bin/coqtop -emacs")
```

to your '`.emacs`' file.

A.4 Notes for syssies

Here are some more notes for installing Proof General in more complex ways. Only attempt things in this section if you really understand what you're doing!

Byte compilation

Compilation of the Emacs lisp files improves efficiency but can sometimes cause compatibility problems, especially if you use more than one version of Emacs with the same `.elc` files.

If you discover problems using the byte-compiled `.elc` files which aren't present using the source `.el` files, please report them to us.

You can compile Proof General by typing `make` in the directory where you installed it. It may be necessary to do this if you use a different version of Emacs.

Site-wide installation

If you are installing Proof General site-wide, you can put the components in the standard directories of the filesystem if you prefer, providing the variables in '`proof-site.el`' are adjusted accordingly (see *Proof General site configuration* in *Adapting Proof General* for more details). Make sure that the '`generic/`' and assistant-specific elisp files are kept in subdirectories ('`coq/`', '`isa/`', '`lego/`') of `proof-home-directory` so that the autoload directory calculations are correct.

To prevent every user needing to edit their own '`.emacs`' files, you can put the `load-file` command to load '`proof-site.el`' into '`site-start.el`' or similar. Consult the Emacs documentation for more details if you don't know where to find this file.

Removing support for unwanted provers

You cannot run more than one instance of Proof General at a time: so if you're using Coq, visiting an '`.ML`' file will not load Isabelle Proof General, and the buffer remains in fundamental mode. If there are some assistants supported that you never want to use, you can adjust the variable `proof-assistants` in '`proof-site.el`' to remove the extra autoloads. This is advisable in case the extensions clash with other Emacs modes, for example Verilog mode for '`.v`' files clashes with Coq mode.

See *Proof General site configuration* in *Adapting Proof General*, for more details of how to adjust the `proof-assistants` setting.

Instead of altering `proof-assistants`, a simple way to disable support for some prover is to delete the relevant directories from the PG installation. For example, to remove support for Coq, delete the '`coq`' directory in the Proof General home directory.

Appendix B Bugs and Enhancements

For an up-to-date description of bugs and other issues, please consult the bugs file included in the distribution: 'BUGS'.

If you discover a problem which isn't mentioned in 'BUGS', please use the search facility on our Trac tracking system at http://proofgeneral.inf.ed.ac.uk/trac. If you cannot find the problem mentioned, please add a ticket, giving a careful description of how to repeat your problem, and saying **exactly** which versions of all Emacs and theorem prover you are using.

If you have some suggested enhancements to request or contribute, please also use the tracking system at http://proofgeneral.inf.ed.ac.uk/trac for this.

References

A short overview of the Proof General system is described in the note:

- [**Asp00**] David Aspinall. *Proof General: A Generic Tool for Proof Development*. Tools and Algorithms for the Construction and Analysis of Systems, Proc TACAS 2000. LNCS 1785.

Script management as used in Proof General is described in the paper:

- [**BT98**] Yves Bertot and Laurent Thry. *A generic approach to building user interfaces for theorem provers*. Journal of Symbolic Computation, 25(7), pp. 161-194, February 1998.

Proof General has support for proof by pointing, as described in the document:

- [**BKS97**] Yves Bertot, Thomas Kleymann-Schreiber and Dilip Sequeira. *Implementing Proof by Pointing without a Structure Editor*. LFCS Technical Report ECS-LFCS-97-368. Also published as Rapport de recherche de l'INRIA Sophia Antipolis RR-3286

History of Proof General

It all started some time in 1994. There was no Emacs interface for LEGO. Back then, Emacs militants worked directly with the Emacs shell to interact with the LEGO system.

David Aspinall convinced Thomas Kleymann that programming in Emacs Lisp wasn't so difficult after all. In fact, Aspinall had already implemented an Emacs interface for Isabelle with bells and whistles, called Isamode. Soon after, the package `lego-mode` was born. Users were able to develop proof scripts in one buffer. Support was provided to automatically send parts of the script to the proof process. The last official version with the name `lego-mode` (1.9) was released in May 1995.

The interface project really took off the ground in November 1996. Yves Bertot had been working on a sophisticated user interface for the Coq system (CtCoq) based on the generic environment Centaur. He visited the Edinburgh LEGO group for a week to transfer proof-by-pointing technology. Even though proof-by-pointing is an inherently structure-conscious algorithm, within a week, Yves Bertot, Dilip Sequeira and Thomas Kleymann managed to implement a first prototype of proof-by-pointing in the Emacs interface for LEGO [BKS97].

Perhaps we could reuse even more of the CtCoq system. It being a structure editor did no longer seem to be such an obstacle. Moreover, to conveniently use proof-by-pointing in actual developments, one would need better support for script management.

In 1997, Dilip Sequeira implemented script management in our Emacs interface for LEGO following the recipe in [BT98]. Inspired by the project CROAP, the implementation made some effort to be generic. A working prototype was demonstrated at UITP'97.

In October 1997, Healfdene Goguen ported `lego-mode` to Coq. Part of the generic code in the `lego` package was outsourced (and made more generic) in a new package called `proof`. Dilip Sequeira provided some LEGO-specific support for handling multiple files and wrote a few manual pages. The system was reasonably robust and we shipped out the package to friends.

In June 1998, David Aspinall reentered the picture by providing an instantiation for Isabelle. Actually, our previous version wasn't quite as generic as we had hoped. Whereas LEGO and Coq are similar systems in many ways, Isabelle was really a different beast. Fierce re-engineering and various usability improvements were provided by Aspinall and Kleymann to make it easier to instantiate to new proof systems. The major technical improvement was a truly generic extension of script management to work across multiple files.

It was time to come up with a better name than just `proof` mode. David Aspinall suggested *Proof General* and set about reorganizing the file structure to disentangle the Proof General project from LEGO at last. He cooked up some images and bolted on a toolbar, so a naive user can replay proofs without knowing a proof assistant language or even Emacs hot-keys. He also designed some web pages, and wrote most of this manual.

Despite views of some detractors, we demonstrated that an interface both friendly and powerful can be built on top of Emacs. Proof General 2.0 was the first official release of the improved program, made in December 1998.

Version 2.1 was released in August 1999. It was used at the Types Summer School held in Giens, France in September 1999 (see `http://www-sop.inria.fr/types-project/types-sum-school.html`). About 50 students learning Coq, Isabelle, and LEGO used Proof General for all three systems. This experience provided invaluable feedback and encouragement to make the improvements that went into Proof General 3.0.

Old News for 3.0

Proof General 3.0 (released November 1999) has many improvements over 2.x releases.

First, there are usability improvements. The toolbar was somewhat impoverished before. It now has twice as many buttons, and includes all of the useful functions used during proof which were previously hidden on the menu, or even only available as key-presses. Key-bindings have been re-organized, users of previous versions may notice. The menu has been redesigned and coordinated with the toolbar, and now gives easy access to more of the features of Proof General. Previously several features were only likely to be discovered by those keen enough to read this manual!

Second, there are improvements, extensions, and bug fixes in the generic basis. Proofs which are unfinished and not explicitly closed by a "save" type command are supported by the core, if they are allowed by the prover. The design of switching the active scripting buffer has been streamlined. The management of the queue of commands waiting to be sent to the shell has been improved, so there are fewer unnecessary `"Proof Process Busy!"` messages. The support for scripting with multiple files was improved so that it behaves reliably with Isabelle99; file reading messages can be communicated in both directions now. The proof shell filter has been optimized to give hungry proof assistants a better share of CPU cycles. Proof-by-pointing has been resurrected; even though LEGO's implementation is incomplete, it seems worth maintaining the code in Proof General so that the implementors of other proof assistants are encouraged to provide support. For one example, we can certainly hope for support in Coq, since the CtCoq proof-by-pointing code has been moved into the Coq kernel lately. We need a volunteer from the Coq community to help to do this.

An important new feature in Proof General 3.0 is support for X-Symbol, which means that real logical symbols, Greek letters, etc can be displayed during proof development, instead of their ASCII approximations. This makes Proof General a more serious competitor to native graphical user interfaces.

Finally, Proof General has become much easier to adapt to new provers — it fails gracefully (or not at all!) when particular configuration variables are unset, and provides more default settings which work out-of-the-box. An example configuration for Isabelle is provided, which uses just 25 or so simple settings.

This manual has been updated and extended for Proof General 3.0. Amongst other improvements, it has a better description of how to add support for a new prover.

See the `CHANGES` file in the distribution for more information about the latest improvements in Proof General. Developers should check the `ChangeLog` in the developer's release for detailed comments on internal changes.

Most of the work for Proof General 3.0 has been done by David Aspinall. Markus Wenzel helped with Isabelle support, and provided invaluable feedback and testing, especially for the improvements to multiple file handling. Pierre Courtieu took responsibility from Patrick Loiseleur for Coq support, although improvements in both Coq and LEGO instances for this release were made by David Aspinall. Markus Wenzel provided support for his Isar language, a new proof language for Isabelle. David von Oheimb helped to develop the generic version of his X-Symbol addition which he originally provided for Isabelle.

A new instantiation of Proof General is being worked on for *Plastic*, a proof assistant being developed at the University of Durham.

Old News for 3.1

Proof General 3.1 (released March 2000) is a bug-fix improvement over version 3.0. There are some minor cosmetic improvements, but large changes have been held back to ensure stability. This release solves a few minor problems which came to light since the final testing stages for

3.0. It also solves some compatibility problems, so now it works with various versions of Emacs which we hadn't tested with before (non-mule GNU Emacs, certain Japanese Emacs versions).

We're also pleased to announce HOL Proof General, a new instance of Proof General for HOL98. This is supplied as a "technology demonstration" for HOL users in the hope that somebody from the HOL community will volunteer to adopt it and become a maintainer and developer. (Otherwise, work on HOL Proof General will not continue).

Apart from that there are a few other small improvements. Check the CHANGES file in the distribution for full details.

The HOL98 support and much of the work on Proof General 3.1 was undertaken by David Aspinall while he was visiting ETL, Osaka, Japan, supported by the British Council and ETL.

Old News for 3.2

Proof General 3.2 introduced several new features and some bug fixes. One noticeable new feature is the addition of a prover-specific menu for each of the supported provers. This menu has a "favourites" feature that you can use to easily define new functions. Please contribute other useful functions (or suggestions) for things you would like to appear on these menus.

Because of the new menus and to make room for more commands, we have made a new key map for prover specific functions. These now all begin with `C-c C-a`. This has changed a few key bindings slightly.

Another new feature is the addition of prover-specific completion tables, to encourage the use of Emacs's completion facility, using `C-RET`. See Section 5.4 [Support for completion], page 32, for full details.

A less obvious new feature is support for turning the proof assistant output on and off internally, to improve efficiency when processing large scripts. This means that more of your CPU cycles can be spent on proving theorems.

Adapting for new proof assistants continues to be made more flexible, and easier in several places. This has been motivated by adding experimental support for some new systems. One new system which had good support added in a very short space of time is **PhoX** (see the PhoX home page for more information). PhoX joins the rank of officially supported Proof General instances, thanks to its developer Christophe Raffalli.

Breaking the manual into two pieces was overdue: now all details on adapting Proof General, and notes on its internals, are in the *Adapting Proof General* manual. You should find a copy of that second manual close to wherever you found this one; consult the Proof General home page if in doubt.

The internal code of Proof General has been significantly overhauled for this version, which should make it more robust and readable. The generic code has an improved file structure, and there is support for automatic generation of autoload functions. There is also a new mechanism for defining prover-specific customization and instantiation settings which fits better with the customize library. These settings are named in the form `PA-setting-name` in the documentation; you replace *PA* by the symbol for the proof assistant you are interested in. See Chapter 7 [Customizing Proof General], page 37, for details.

Finally, important bug fixes include the robustification against `write-file` (`C-x C-w`), `revert-buffer`, and friends. These are rather devious functions to use during script management, but Proof General now tries to do the right thing if you're deviant enough to try them out!

Work on this release was undertaken by David Aspinall between May-September 2000, and includes contributions from Markus Wenzel, Pierre Courtieu, and Christophe Raffalli. Markus added some Isar documentation to this manual.

Old News for 3.3

Proof General 3.3 includes a few feature additions, but mainly the focus has been on compatibility improvements for new versions of provers (in particular, Coq 7), and new versions of emacs (in particular, XEmacs 21.4).

One new feature is control over visibility of completed proofs, See Section 3.3 [Visibility of completed proofs], page 20. Another new feature is the tracking of theorem dependencies inside Isabelle. A context-sensitive menu (right-button on proof scripts) provides facility for browsing the ancestors and child theorems of a theorem, and highlighting them. The idea of this feature is that it can help you untangle and rearrange big proof scripts, by seeing which parts are interdependent. The implementation is provisional and not documented yet in the body of this manual. It only works for the "classic" version of Isabelle99-2.

Old News for 3.4

Proof General 3.4 adds improvements and also compatibility fixes for new versions of Emacs, in particular, for GNU Emacs 21, which adds the remaining pretty features that have only been available to XEmacs users until now (the toolbar and X-Symbol support).

One major improvement has been to provide better support for synchronization with Coq proof scripts; now Coq Proof General should be able to retract and replay most Coq proof scripts reliably. Credit is due to Pierre Courtieu, who also updated the documentation in this manual.

As of version 3.4, Proof General is distributed under the GNU General Public License (GPL). Compared with the previous more restrictive license, this means the program can now be redistributed by third parties, and used in any context without applying for a special license. Despite these legal changes, we would still appreciate if you send us back any useful improvements you make to Proof General, and register your use of Proof General on the web site.

Old News for 3.5

Old News for 3.6

There was no 3.6 release of Proof General.

Old News for 3.7

Proof General version 3.7.1 is an updated and enhanced version of Proof General 3.7. See '`CHANGES`' for more details.

Proof General version 3.7 collects together a cummulative set of improvements to Proof General 3.5. There are compatibility fixes for newer Emacs versions, and particularly for GNU Emacs: credit is due to Stefan Monnier for an intense period of debugging and patching. The options menu has been simplified and extended, and the display management is improved and repaired for Emacs API changes. There are some other usability improvements, some after feedback from use at TYPES Summer Schools. Many new features have been added to enhance Coq mode (thanks to Pierre Courtieu) and several improvements made for Isabelle (thanks to Makarius Wenzel, Stefan Berghofer and Tjark Weber).

Support has been added for the useful Emacs packages Speedbar and Index Menu, both usually distributed with Emacs. Compatible versions of the Emacs packages Math-Menu (for Unicode symbols) and MMM Mode (for multiple modes in one buffer) are bundled with Proof General. An experimental Unicode Tokens package has been added which will replace X-Symbol.

See the '`CHANGES`' file in the distribution for more complete details of changes since version 3.5, and the appendix [History of Proof General], page 67 for old news.

Function and Command Index

A

add-completions-from-tags-table 33

C

complete 32

I

indent-for-tab-command 12
isabelle-choose-logic 53
isar-strip-terminators 55

P

pg-goals-button-action 35
pg-hide-all-proofs 21
pg-identifier-under-mouse-query 36
pg-next-input 23
pg-next-matching-input 24
pg-next-matching-input-from-input 24
pg-previous-input 23
pg-previous-matching-input 23
pg-previous-matching-input-from-input 24
pg-response-clear-displays 15
pg-show-all-proofs 21
pg-toggle-visibility 20
proof-active-area-face 43
proof-assert-next-command-interactive 13
proof-assert-until-point-interactive 14
proof-autosend-toggle 20
proof-boring-face 43
proof-ctxt 15
proof-debug-message-face 43
proof-declaration-name-face 43
proof-display-some-buffers 14, 39
proof-eager-annotation-face 43
proof-electric-terminator-toggle 14
proof-error-face 43
proof-find-theorems 15
proof-frob-locked-end 23
proof-goto-command-end 12
proof-goto-command-start 12
proof-goto-end-of-locked 12
proof-goto-point 13
proof-help 15
proof-highlight-dependency-face 42
proof-highlight-dependent-face 42
proof-interrupt-process 15
proof-issue-goal 16
proof-issue-save 16
proof-layout-windows 39
proof-locked-face 42
proof-minibuffer-cmd 15
proof-mouse-highlight-face 42
proof-prf 14
proof-process-buffer 13
proof-query-identifier 15
proof-queue-face 42
proof-retract-buffer 14
proof-retract-until-point-interactive 14
proof-script-highlight-error-face 42
proof-script-sticky-error-face 42
proof-shell-exit 16
proof-shell-restart 16
proof-shell-start 16
proof-tacticals-name-face 43
proof-toggle-active-scripting 11
proof-undo-and-delete-last-successful-command 13
proof-undo-last-successful-command 13
proof-warning-face 43

U

unicode-tokens-copy 28
unicode-tokens-fraktur-font-face 27
unicode-tokens-list-shortcuts 28
unicode-tokens-list-tokens 28
unicode-tokens-list-unicode-chars 28
unicode-tokens-paste 28
unicode-tokens-sans-font-face 27
unicode-tokens-script-font-face 27
unicode-tokens-serif-font-face 27
unicode-tokens-symbol-font-face 29

Variable and User Option Index

C

coq-mode-hooks 31

I

isa-mode-hooks 31
isabelle-chosen-logic 53
isabelle-program-name-override 53
isabelle-web-page 55

L

lego-mode-hooks 31
lego-tags 49
lego-www-home-page 49

P

PA-completion-table 32
PA-one-command-per-line 40
PA-script-indent 40
pg-input-ring-size 41
proof-assistant-home-page 43
proof-auto-action-when-deactivating-scripting 41
proof-auto-raise-buffers 38
proof-autosend-enable 20
proof-colour-locked 39
proof-delete-empty-windows 39
proof-disappearing-proofs 21
proof-electric-terminator-enable 40
proof-follow-mode 41
proof-general-debug 41
proof-goal-with-hole-regexp 31
proof-goal-with-hole-result 31
proof-keep-response-history 41
proof-multiple-frames-enable 39
proof-prog-name-ask 41
proof-prog-name-guess 41
proof-query-file-save-when-activating-scripting 40
proof-rsh-command 42
proof-script-indent 12
proof-shrink-windows-tofit 39
proof-splash-enable 40
proof-strict-read-only 40
proof-terminal-string 12
proof-three-window-enable 38
proof-tidy-response 41
proof-toolbar-enable 40

U

unicode-tokens-font-family-alternatives 29
unicode-tokens-highlight-unicode 28

Keystroke Index

C

C-c C- 12
C-c C-a 12
C-c C-a b 55
C-c C-a C-a 51, 55
C-c C-a C-b 55
C-c C-a C-d 54
C-c C-a C-e 51
C-c C-a C-f 54
C-c C-a C-i 49, 51, 55
C-c C-a C-I 49
C-c C-a C-l 55
C-c C-a C-m 54
C-c C-a C-o 51
C-c C-a C-p 54
C-c C-a C-q 54
C-c C-a C-r 55
C-c C-a C-R 49
C-c C-a C-s 51, 54
C-c C-a C-u 55
C-c C-a h a 54
C-c C-a h A 54
C-c C-a h b 54
C-c C-a h c 54
C-c C-a h C 54
C-c C-a h f 54
C-c C-a h i 54
C-c C-a h I 54
C-c C-a h m 54
C-c C-a h o 54
C-c C-a h S 54
C-c C-a h t 54
C-c C-a h T 54
C-c C-a l 55
C-c C-a r 54
C-c C-a u 55
C-c C-b 13
C-c C-BS 13
C-c C-c 14
C-c C-e 12
C-c C-f 14, 54
C-c C-h 14
C-c C-n 13
C-c C-p 14
C-c C-r 13
C-c C-RET 13
C-c C-t 14
C-c C-u 13
C-c C-v 14

M

M-n 23
M-p 23

Concept Index

A

active scripting buffer ... 11
Alt ... 5
annotation ... 19
Assertion ... 10, 22
auto raise ... 38
Automatic processing ... 20
autosend ... 20

B

blue text ... 10
buffer display customization ... 38

C

Centaur ... 67
colour ... 31
completion ... 32
CtCoq ... 67
Customization ... 37

D

Dedicated windows ... 40
display customization ... 38

E

Editing region ... 10
Emacs customization library ... 37

F

Features ... 4
file variables ... 46
font lock ... 31
frames ... 38
Future ... 1

G

generic ... 67
goal ... 10
goal-save sequences ... 10
goals buffer ... 11
Greek letters ... 25

H

history ... 67
HOL Proof General ... 57

I

Imenu ... 31
Indentation ... 40
index menu ... 31
Input ring ... 23, 40
Isabelle commands ... 54
Isabelle customizations ... 55
Isabelle logic ... 53
Isabelle Proof General ... 53

K

key sequences ... 5
keybindings ... 45

L

LEGO Proof General ... 49
`lego-mode` ... 1, 67
Locked region ... 10
logical symbols ... 25

M

maintenance ... 1
mathematical symbols ... 25
Maths Menu ... 25
Meta ... 5
Multiple Files ... 19
multiple frames ... 38
multiple windows ... 38

N

news ... 1, 68, 69

O

outline mode ... 32

P

pink text ... 10
prefix argument ... 13
proof assistant ... 3
proof by pointing ... 11, 67
Proof General ... 3
Proof General Kit ... 1
proof script ... 9
Proof script indentation ... 40
proof script mode ... 10

Q

Query program name ... 40
Queue region ... 10

R

Remote host ... 40
Remote shell ... 40
response buffer ... 11
Retraction ... 10, 22
Running proof assistant remotely ... 40

S

save ... 10
script buffer ... 10
script management ... 67
scripting ... 9
Shell ... 22
shell buffer ... 11
Shell Proof General ... 59
Speedbar ... 31
Strict read-only ... 40
structure editor ... 67
subscripts ... 25
superscripts ... 25
Switching between proof scripts ... 21
symbols ... 25

T

tags ... 33
three-buffer interaction ... 38
Tokens Mode ... 25
Toolbar button enablers ... 40
Toolbar disabling ... 40
Toolbar follow mode ... 40

U

Undo in read-only region ... 40
User options ... 40
Using Customize ... 37

V

Visibility of proofs ... 20

W

Why use Proof General? ... 4

X

X-Symbols ... 25

Table of Contents

Preface ... 1
News for Version 4.0 ... 1
Future ... 1
Credits ... 1

1 Introducing Proof General ... 3
1.1 Installing Proof General ... 3
1.2 Quick start guide ... 3
1.3 Features of Proof General ... 4
1.4 Supported proof assistants ... 5
1.5 Prerequisites for this manual ... 5
1.6 Organization of this manual ... 6

2 Basic Script Management ... 7
2.1 Walkthrough example in Isabelle ... 7
2.2 Proof scripts ... 9
2.3 Script buffers ... 10
2.3.1 Locked, queue, and editing regions ... 10
2.3.2 Goal-save sequences ... 10
2.3.3 Active scripting buffer ... 11
2.4 Summary of Proof General buffers ... 11
2.5 Script editing commands ... 12
2.6 Script processing commands ... 13
2.7 Proof assistant commands ... 14
2.8 Toolbar commands ... 16
2.9 Interrupting during trace output ... 16

3 Advanced Script Management and Editing ... 19
3.1 Document centred working ... 19
3.2 Automatic processing ... 20
3.3 Visibility of completed proofs ... 20
3.4 Switching between proof scripts ... 21
3.5 View of processed files ... 21
3.6 Retracting across files ... 22
3.7 Asserting across files ... 22
3.8 Automatic multiple file handling ... 22
3.9 Escaping script management ... 22
3.10 Editing features ... 23

4 Unicode symbols and special layout support ... 25
4.1 Maths menu ... 25
4.2 Unicode Tokens mode ... 25
4.3 Configuring tokens symbols and shortcuts ... 26
4.4 Special layout ... 26
4.5 Moving between Unicode and tokens ... 27
4.6 Finding available tokens shortcuts and symbols ... 28
4.7 Selecting suitable fonts ... 29

5 Support for other Packages 31
5.1 Syntax highlighting 31
5.2 Imenu and Speedbar 31
5.3 Support for outline mode 32
5.4 Support for completion 32
5.5 Support for tags 33

6 Subterm Activation and Proof by Pointing 35
6.1 Goals buffer commands 35

7 Customizing Proof General 37
7.1 Basic options 37
7.2 How to customize 37
7.3 Display customization 38
7.4 User options 40
7.5 Changing faces 42
7.5.1 Script buffer faces 42
7.5.2 Goals and response faces 43
7.6 Tweaking configuration settings 43

8 Hints and Tips 45
8.1 Adding your own keybindings 45
8.2 Using file variables 46
8.3 Using abbreviations 46

9 LEGO Proof General 49
9.1 LEGO specific commands 49
9.2 LEGO tags 49
9.3 LEGO customizations 49

10 Coq Proof General 51
10.1 Coq-specific commands 51
10.2 Editing multiple proofs 51
10.3 User-loaded tactics 51
10.4 Holes feature 52

11 Isabelle Proof General 53
11.1 Choosing logic and starting isabelle 53
11.2 Isabelle commands 54
11.3 Isabelle settings 55
11.4 Isabelle customizations 55

12 HOL Proof General 57

13 Shell Proof General 59

Appendix A Obtaining and Installing **61**
A.1 Obtaining Proof General 61
A.2 Installing Proof General from tarball 61
A.3 Setting the names of binaries 61
A.4 Notes for syssies 62
Byte compilation 62
Site-wide installation 62
Removing support for unwanted provers 62

Appendix B Bugs and Enhancements **63**

References **65**

History of Proof General **67**
Old News for 3.0 68
Old News for 3.1 68
Old News for 3.2 69
Old News for 3.3 70
Old News for 3.4 70
Old News for 3.5 70
Old News for 3.6 70
Old News for 3.7 70

Function and Command Index **71**

Variable and User Option Index **73**

Keystroke Index **75**

Concept Index **77**